Aiello, L. C., Nardi, D. & Schaerf, M.	REASONING ABOUT KNOWLEDGE
Anderson, J. (editor)	POP-11 COMES OF AGE: The Advancement of an AI Programming Language
Andrew, A. M.	CONTINUOUS HEURISTICS: The Prelinguistic Basis of Intelligence
Al-Attar, A.	KNOWLEDGE ENGINEERING
Beal, D. F.	ADVANCES IN COMPUTER CHESS 6
Bergadano, F., Giordana, A. & Saitta, L.	MACHINE LEARNING: An Integrated Framework and Its Applications
Bestougeff, H. & Ligozat, G.	LOGICAL TOOLS FOR TEMPORAL KNOWLEDGE REPRESENTATION
Bläsius, K. H. & Bürckert, H.-J.	DEDUCTION SYSTEMS IN AI
Campbell, J. A. (editor)	IMPLEMENTATIONS OF PROLOG
Campbell, J. A. & Cuena, J. (editors)	PERSPECTIVES IN ARTIFICIAL INTELLIGENCE, Vols 1 & 2
Carter, D.	INTERPRETING ANAPHORS IN NATURAL LANGUAGE TEXTS
Craig, I. D.	FORMAL SPECIFICATION OF ADVANCED AI ARCHITECTURES
Davies, R. (editor)	INTELLIGENT INFORMATION SYSTEMS: Progress and Prospects
Evans, J. B.	STRUCTURES OF DISCRETE EVENT SIMULATION
Farreny, H.	AI AND EXPERTISE
Forsyth, R. & Rada, R.	MACHINE LEARNING: Applications in Expert Systems and Information Retrieval
Frixione, S. G., Gaglio, S. & Spinelli, G.	REPRESENTING CONCEPTS IN SEMANTIC NETS
Fūtó, I. & Gergely, T.	ARTIFICIAL INTELLIGENCE SIMULATION
Gabbay, D. M.	ELEMENTARY LOGICS: A Procedural Perspective*
Gardin, J.-C.	ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS: Case Studies in the Knowledge Domain of Archaeology
Gottinger, H. W. & Weimann, H. P.	ARTIFICIAL INTELLIGENCE: A Tool for Industry and Management
Gray, P. M. D. & Lucas, R. J.	PROLOG AND DATABASES: Implementations and New Directions
Hawley, R. (editor)	ARTIFICIAL INTELLIGENCE PROGRAMMING ENVIRONMENTS
Herik, J. van der & Allis, V.	HEURISTIC PROGRAMMING IN ARTIFICIAL INTELLIGENCE 3: The Third Computer Olympiad
Hodgson, J. P. E.	KNOWLEDGE REPRESENTATION AND LANGUAGE IN AI
Hoek, W. van der, Meyer, J.-J. Ch., Tan, Y. H. & Witteveen, C.	NON-MONOTONIC REASONING & PARTIAL SEMANTICS
Kopec, D. & Thompson, R. B.	ARTIFICIAL INTELLIGENCE AND INTELLIGENT TUTORING SYSTEMS: Knowledge-Based Systems for Teaching and Learning
Levy, D. N. L. & Beal, D. F. (editors)	HEURISTIC PROGRAMMING IN ARTIFICIAL INTELLIGENCE: The First Computer Olympiad
Levy, D. N. L. & Beal, D. F. (editors)	HEURISTIC PROGRAMMING IN ARTIFICIAL INTELLIGENCE 2: The Second Computer Olympiad
Lopez de Mantaras, R.	APPROXIMATE REASONING MODELS
Lukaszewicz, W.	NON-MONOTONIC REASONING: Formalization of Commonsense Reasoning
McGraw, K. & Westphal, C.	READINGS IN KNOWLEDGE ACQUISITION: Current Practices and Trends
Mellish, C.	COMPUTER INTERPRETATION OF NATURAL LANGUAGE DESCRIPTIONS
Michie, D.	ON MACHINE INTELLIGENCE, Second Edition
Mortimer, H.	THE LOGIC OF INDUCTION
Obermeier, K. K.	NATURAL LANGUAGE PROCESSING TECHNOLOGIES IN ARTIFICIAL INTELLIGENCE
Partridge, D.	ARTIFICIAL INTELLIGENCE: Applications in the Future of Software Engineering
Ramsay, A. & Barrett, R.	AI IN PRACTICE: Examples in POP-11
Ras, Z. W. & Zemankova, M.	INTELLIGENT SYSTEMS: State of the Art and Future Directions
Saint-Dizier, P. & Szpakowicz, S. (editors)	LOGIC AND LOGIC GRAMMARS FOR LANGUAGE PROCESSING
Savory, S. E.	ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS
Savory, S. E.	EXPERT SYSTEMS IN THE ORGANISATION
Shanahan, M. & Southwick, R.	SEARCH, INFERENCE AND DEPENDENCIES IN ARTIFICIAL INTELLIGENCE
Sparck Jones, K. & Wilks, Y. (editors)	AUTOMATIC NATURAL LANGUAGE PARSING
Smith, B. & Kelleher, G. (editors)	REASON MAINTENANCE SYSTEMS AND THEIR APPLICATIONS
Thanassas, D.	THE PHENOMENON OF COMMONSENSE REASONING: Nonmonotonicity, Action and Information
Torrance, S. (editor)	THE MIND AND THE MACHINE
Turner, R.	LOGICS FOR ARTIFICIAL INTELLIGENCE
Vernon, D. & Sandini, G.	PARALLEL COMPUTER VISION: The VIS-a-VIS System
Wallace, M.	COMMUNICATING WITH DATABASES IN NATURAL LANGUAGE
Wertz, H.	AUTOMATIC CORRECTION AND IMPROVEMENT OF PROGRAMS
Yazdani, M. (editor)	NEW HORIZONS IN EDUCATIONAL COMPUTING
Yazdani, M. & Narayanan, A. (editors)	ARTIFICIAL INTELLIGENCE: Human Effects
Zeidenberg, M.	NEURAL NETWORKS IN ARTIFICIAL INTELLIGENCE

* *in preparation*

MASSIVELY PARALLEL MODELS OF COMPUTATION
Distributed Parallel Processing in Artificial Intelligence and Optimisation

DR VALMIR C BARBOSA
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil

ELLIS HORWOOD
NEW YORK LONDON TORONTO SYDNEY TOKYO SINGAPORE

VISIT…

LANZAROTE
Caliente.COM

First published in 1993 by
Ellis Horwood Limited
Campus 400, Maylands Avenue
Hemel Hempstead
Hertfordshire, HP2 7EZ
and
Market Cross House, Cooper Street
Chichester, West Sussex PO19 1EB, England
A division of
Simon & Schuster International Group

Printed and bound in Great Britain
by Bookcraft, Midsomer Norton.

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

ISBN 0-13-562968-3 (hbk)

Library of Congress Cataloging-in-Publication Data
Available from the publisher.

2 3 4 5 97 96 95 94

To Alzira, Leonardo, Julia, and Isabel

Contents

Preface **xi**

Part 1. Introduction and background **1**

1. Introduction **3**

1.1. Preliminaries 3
1.2. Two classes of automaton networks 4
1.3. Bibliographic notes 5

2. Background **9**

2.1. Graphs 9
2.2. Complexity 11
2.3. Probabilities and stochastic processes 13
2.4. Bibliographic notes 16

Part 2. Fundamentals of distributed parallel computation **19**

3. Models of distributed parallel computation **21**

3.1. Distributed algorithms 21
3.1.1. The nature of distributed algorithms 21
3.1.2. Models of distributed parallel computation 23
3.1.3. Some formalism 25
3.1.4. Complexity measures 28
3.1.5. Notation 30
3.2. Distributed snapshots 33
3.2.1. Global states of a distributed computation 33
3.2.2. An algorithm for global state recording 37
3.3. Termination detection 41
3.3.1. The general case 41
3.3.2. The case of diffusing computations 43
3.4. Bibliographic notes 48

4. Timing and synchronization **49**
4.1. Synchronizers 49
4.1.1. Synchronizing algorithms 49
4.1.2. Some synchronizers 51
4.2. Scheduling by edge reversal 56
4.2.1. Neighborhood constraints 56
4.2.2. The algorithm 57
4.2.3. Some properties 61
4.3. Template algorithms 66
4.3.1. A generic architecture 66
4.3.2. Fully concurrent automaton networks 67
4.3.3. Partially concurrent automaton networks 69
4.4. Bibliographic notes 71

Part 3. Fully concurrent automaton networks **73**

5. Cellular automata **75**
5.1. The model and basic properties 75
5.2. Simulation algorithms 79
5.2.1. The sequential algorithm 79
5.2.2. The distributed parallel algorithm 81
5.3. Further properties in the one-dimensional case 82
5.4. Introducing probabilistic updating functions 85
5.5. Bibliographic notes 87

6. Analog Hopfield neural networks **89**
6.1. The model and basic properties 89
6.2. Simulation algorithms 92
6.2.1. The sequential algorithm 92
6.2.2. The distributed parallel algorithm 95
6.3. The traveling salesman problem 96
6.4. Bibliographic notes 101

7. Other analog neural networks **103**
7.1. Systems of linear algebraic equations 103
7.2. Simulation algorithms 108
7.2.1. The sequential algorithm 108
7.2.2. The distributed parallel algorithm 110
7.3. Linear programming 111
7.4. Bibliographic notes 114

Part 4. Partially concurrent automaton networks **117**

8. Binary Hopfield neural networks **119**
8.1. The model and basic properties 119
8.2. Simulation algorithms 122

8.2.1. The sequential algorithm 122
8.2.2. The distributed parallel algorithm 124
8.3. The network as an associative memory 125
8.4. The minimum node cover problem 129
8.5. Bibliographic notes 133

9. Markov random fields **135**
9.1. The model and basic properties 135
9.2. Simulation algorithms 138
9.2.1. The sequential algorithm 138
9.2.2. The distributed parallel algorithm 139
9.3. Further properties 140
9.3.1. Simulated annealing 140
9.3.2. Gibbs sampling and simulated annealing 143
9.4. Further simulation algorithms 144
9.4.1. The sequential algorithm 144
9.4.2. The distributed parallel algorithm 145
9.5. Yet another property 146
9.6. Boltzmann machines 147
9.6.1. The model and basic properties 147
9.6.2. The minimum node cover problem 149
9.7. Bibliographic notes 149

10. Bayesian networks **153**
10.1. The model and basic properties 153
10.2. Simulation algorithms 159
10.2.1. The sequential algorithm 159
10.2.2. The distributed parallel algorithm 161
10.2.3. Two variations of stochastic simulation 163
10.3. Further properties 163
10.4. Further simulation algorithms 164
10.4.1. The sequential algorithm 164
10.4.2. The distributed parallel algorithm 166
10.5. Lexical disambiguation 168
10.6. Bibliographic notes 175

Part 5. Appendices **177**

A. A distributed parallel programming primer **179**
A.1. Distributed parallel programs 179
A.2. Message buffering 182
A.3. Task allocation 187
A.4. Interprocessor communication 196
A.5. Bibliographic notes 198

B. The software for automaton network simulation **201**
B.1. Occam programming 201
B.2. Occam programs from distributed algorithms 206
B.3. Automaton network simulation in Occam 209
B.4. Bibliographic notes 209

C. Long proofs **211**
C.1. Proof of Theorem 4.4 211
C.2. Proof of Theorem 9.1 216
C.3. Bibliographic notes 221

D. Additional edge-reversal properties **223**
D.1. Reachability equations 223
D.2. Bibliographic notes 224

Bibliography **225**

Author index **239**

Subject index **243**

Preface

This is a book about the parallel simulation by distributed-memory machines of massively parallel models of computation within artificial intelligence and optimization. The models treated include cellular automata, Hopfield neural networks (both analog and binary), Markov random fields, Boltzmann machines, Bayesian networks, and other analog neural networks specialized in the solution of some mathematical problems. I have intended the book to have a multidisciplinary character, so it contains, in addition to the simulation-related material and at different levels of detail, a treatment of basic principles of distributed parallel program design, of each model's main properties and applications, and of how the models relate to one another.

The material contained in this book may appeal to professionals and students in a variety of disciplines, as in computer science (particularly within artificial intelligence, optimization, and distributed parallel processing), electrical engineering, and cognitive science. It may also be of interest as a complementary textbook for senior undergraduates and graduates in these areas.

This book comprises ten chapters and four appendices, grouped into five major parts. Every chapter and appendix is complemented by a section with bibliographic notes, where comments and directions regarding the bibliography section at the end of the book are provided.

Part 1 contains Chapters 1 and 2, and is devoted to introductory and background material. Chapter 1 presents an introduction to automaton networks and a categorization of these networks into the two major classes of interest in the remainder of the book. These are the classes of fully concurrent and partially concurrent automaton networks, defined in terms of the dynamic behavior of the networks. Every model treated in later chapters falls within one of these two classes. Chapter 2 contains a little background on graph theory, complexity of algorithms, probabilities, and stochastic processes. Most readers will be able to skip this chapter, which serves mainly as a quick reference for basic concepts.

Part 2 comprises Chapters 3 and 4, devoted respectively to a discussion of basic algorithmic techniques within a distributed parallel context (Chapter 3), and to the presentation of a generic simulator architecture for fully concurrent and partially concurrent automaton networks (Chapter 4). The material in Chapter 3 is complemented by Appendix A, in which various issues of importance at a less abstract level are addressed. The remaining three appendices complement the material in Chapter 4 with different emphases. Appendix B presents a discussion on how the simulators would be implemented in the Occam language, while Appendices C and D contain additional material on the basic distributed algorithm used to simulate partially concurrent automaton networks (respectively, a long theorem proof and additional properties).

Each of the additional chapters (Chapters 5 through 10) is devoted to one of the models treated in the book, respectively cellular automata, analog Hopfield neural networks, other analog neural networks, binary Hopfield neural networks, Markov random fields (including Boltzmann machines), and Bayesian networks. Chapters 5 through 7 constitute Part 3, and deal with fully concurrent automaton networks, while Part 4, on partially concurrent automaton networks, comprises Chapters 8 through 10. Part 5 comprises Appendices A through D.

Every one of Chapters 5 through 10 contains a discussion of its model's main properties, the algorithms to simulate the model's dynamic behavior, and some of the model's main applications. Chapter 9, on Markov random fields, is complemented by Appendix C, where a long theorem proof is presented. Chapters 5 through 10 may in general be read as self-contained units, although by the very nature of the book there is a lot of cross-referencing among some of them. For a proper understanding of the sections on the distributed parallel simulation of each model, a prior contact with Chapters 1, 3, and 4 is naturally recommended.

Various portions of this book contain results of research over which many of my students and colleagues at COPPE/UFRJ collaborated. I would like to single out the work of Luís Alfredo de Carvalho (especially for the material that culminated in Chapter 7) and Leila Eizirik (for the material that led to Section 10.5). I am also thankful to the staff at the IBM Rio Scientific Center, where this book was written and typeset during my first year as a visiting scientist, for providing such an excellent work environment. Finally, I acknowledge very gladly the patient work of Lúcia Drummond and Helena Leitão, who prepared all the illustrations.

Rio de Janeiro, Brazil
December 1992

V.C.B.

Part 1

Introduction and background

This first part of the book is devoted to the presentation of introductory material on automaton networks, and to the discussion of some of the background to be used throughout the other parts.

Part 1 comprises Chapters 1 and 2. Chapter 1 contains the definition and classification of automaton networks into the two categories of interest in this book (fully concurrent and partially concurrent automaton networks). Chapter 2 contains some of the background of interest to most of the subsequent chapters, mainly on graph theory, complexity theory, and probability theory.

1

Introduction

In this chapter, we first introduce automaton networks (Section 1.1), and then proceed in Section 1.2 to a classification of automaton networks into fully concurrent and partially concurrent automaton networks. Bibliographic notes of relevance to the chapter are given in Section 1.3.

1.1. PRELIMINARIES

An *automaton network* $\mathcal{A}$ is a discrete dynamic system defined by the pair $\mathcal{A} = (G, f)$, where $G = (N, E)$ is an undirected graph of node set N and edge set E, and f is the *updating function*. The function f determines the evolution of each node's state for every node in N as a function of its own current state and of the states of nodes connected to it by edges in E (its *neighbors*). The possible states of a node depend on the particular automaton network at hand, and we leave them unspecified for now.

Automaton networks are generally viewed as a powerful model for a variety of complex systems, which are characterized by the presence of a very large number of elementary components of simple, "localized" behavior, but whose collective properties are very hard to analyze. In view of these characteristics, the simulation by computers of automaton networks is a key element in the understanding of those complex systems, especially if the currently available parallel processing systems can be employed.

In several areas of intellectual investigation, complex systems amenable to a modeling by automaton networks appear, as in biology, computer science, economics, and physics, to name some. Neural networks, constituting one class of such complex systems, have in recent years fostered a great interest among investigators of various disciplines, motivated especially by the networks' ability to "learn" structural properties of a problem's domain, and by the increasing feasibility of utilizing (often analog) VLSI chips in their implementation. One learning algorithm that appears to respond for a great share of the current success of neural networks is the back-propagation learning algorithm, used to "train" networks comprising feed-forward connections only (we do not, however, treat such networks in this book, as their dynamic behavior is generally uninteresting outside the learning phase).

What one usually seeks in the study of automaton networks is the understanding of their transient and long-term global properties that the evolution in time as dictated by the function f entails. This understanding is often impaired by enormous difficulties, as the reader will have the opportunity to note along the various chapters of this book. Such difficulties are in flavor very similar to the ones already known to various researchers in connection with some paradigmatic problems in computer science, as the firing squad problem and the chip firing problem. We mention these two problems because, even apparently disconnected from the models discussed in later chapters, they seem to capture the essence of the difficulties involved. The firing squad problem asks that a special updating function f be devised so that every node in N enters a special state, and that all nodes do so for the first time concomitantly. In the chip firing problem, on the other hand, each member of N is initially assigned a certain number of chips, and then the updating function f says that, if a node has at least as many chips as it has neighbors, then exactly one chip is sent to each of its neighbors. These two problems, the reader will note, illustrate the difficulties involved both in utilizing automaton networks to model complex systems and in analyzing them once the model has been set up.

1.2. TWO CLASSES OF AUTOMATON NETWORKS

Let $n = |N|$, and for $1 \leq i \leq n$ let n_i denote a node in N. The evolution in time of an automaton network proceeds in synchronous pulses, here represented by the integer $s \geq 0$, in such a way that the *state* of a node n_i at pulse $s > 0$ is given by f as a function of the states of n_i and of n_i's neighbors in G at pulse $s - 1$. The *initial state* of n_i corresponds to $s = 0$. The assembled states of the n nodes are referred to as the *state* of the automaton network. Let $x_i(s)$ denote the state of n_i at pulse s and $\mathcal{N}(n_i)$ its set of neighbors. If we let $\mathcal{S}(s)$ be the set of nodes to have states updated at pulse s, then the time-evolution of $\mathcal{A}$ is expressed as the *updating rule*

$$x_i(s) = \begin{cases} f\big(x_j(s-1);\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)\big), & \text{if } n_i \in \mathcal{S}(s); \\ \hfill x_i(s-1), & \text{otherwise,} \end{cases} \tag{1.1}$$

for all $n_i \in N$ and all $s > 0$. In (1.1), the notation $x_j(s-1);\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)$ is meant to indicate a dependency of f on the states at time $s - 1$ of all nodes $n_j \in \{n_i\} \cup \mathcal{N}(n_i)$.

Depending on how the set $\mathcal{S}(s)$ is selected for use in (1.1) at each pulse $s > 0$, it is possible to obtain various *updating schedules*. Each of these schedules defines a class of automaton networks. In this book, we are concerned with the following two updating schedules (correspondingly, two classes of automaton networks), henceforth referred to as Fully Concurrent (FC) and Partially Concurrent (PC) updating schedules (the same denominations will be used to designate the respective classes of automaton networks).

- *FC updating schedule.* In this case, $\mathcal{S}(s) = N$ for all $s > 0$, so the updating rule in (1.1) becomes

$$x_i(s) = f\big(x_j(s-1);\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)\big), \tag{1.2}$$

for all $n_i \in N$ and all $s > 0$. The FC updating schedule starts at the initial states $x_i(0)$ for all $n_i \in N$, and for $s > 0$ lets $x_i(s)$ be given as in (1.2) for all $n_i \in N$.

- *PC updating schedule.* In this case, the concurrency in node updating is restricted: $x_i(s)$ and $x_j(s)$ may be updated at pulse s if and only if n_i and n_j are not neighbors; otherwise, at most one of the two may be updated and the other one maintains the value it had at pulse $s - 1$. Let $\mathcal{I}_1, \mathcal{I}_2, \ldots$ denote independent sets in G (an independent set in a graph is a subset of nodes that contains no neighbors in the graph) such that every node in N appears infinitely often in the sequence $\mathcal{I}_1, \mathcal{I}_2, \ldots$ (this means that there is a constant $K \geq 0$ such that every node appears in at least one of $\mathcal{I}_{K_0}, \mathcal{I}_{K_0+1}, \ldots, \mathcal{I}_{K_0+K}$ for all $K_0 > 0$). Then for $s > 0$ let $\mathcal{S}(s) = \mathcal{I}_s$ and rewrite the updating rule (1.1) as

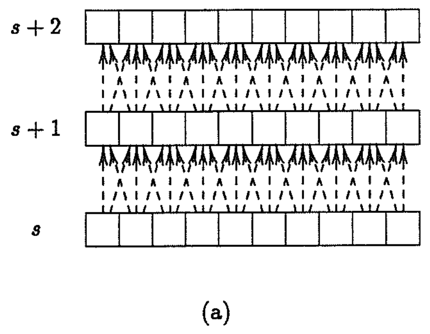
$$x_i(s) = \begin{cases} f\big(x_j(s-1);\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)\big), & \text{if } n_i \in \mathcal{I}_s; \\ \hfill x_i(s-1), & \text{otherwise,} \end{cases} \tag{1.3}$$

for all $n_i \in N$ and all $s > 0$. The PC updating schedule starts at the initial states $x_i(0)$ for all $n_i \in N$, and for $s > 0$ lets $x_i(s)$ be given as in (1.3) for all $n_i \in N$. In other words, at pulse $s = 1$ the nodes in $\mathcal{I}_1$ have their states updated, then at $s = 2$ the nodes in $\mathcal{I}_2$, and so on.

In summary, under the FC updating schedule every node is updated at every pulse $s > 0$, whereas under the PC updating schedule nodes are updated in groups that constitute independent sets in G (Figure 1.1). The latter case depends on the choice of the sets $\mathcal{I}_1, \mathcal{I}_2, \ldots$. One (extreme) possibility is to have all these sets be singletons in such a way that nodes are scanned sequentially and updated one at a time. Depending on the particular phenomenon the automaton network is intended to model, the choice of these independent sets may or may not be crucial. In all cases to be reviewed in this book, however, the sole requirement will be that nodes be updated infinitely often, so we are completely free to choose the arrangement of the independent sets, provided we obey the constraint that every node appears infinitely often in the sequence of independent sets.

1.3. BIBLIOGRAPHIC NOTES

There are various references that complement the material presented in Section 1.1. Fox and Otto (1986) provide further motivation for the involvement of parallel computing in the study of complex systems. Goles and Martínez (1990) treat various types of automaton networks, while specific books on neural networks include those



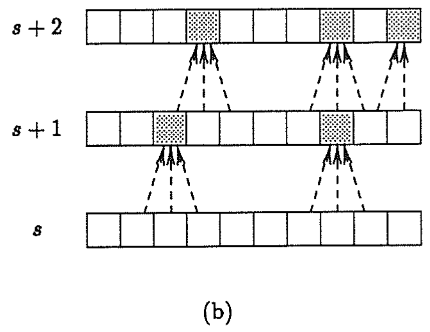


Figure 1.1. *The automaton network considered for parts (a) and (b) is such that G is a line of ten nodes. Two updating cycles are shown for some $s \geq 0$, one at pulse $s+1$ (based on pulse s), the other at pulse $s+2$ (based on pulse $s+1$). All nodes of an FC automaton network have their states updated at both pulses (a). In a PC automaton network, on the other hand, only nodes that are not neighbors are allowed to have their states updated at the same pulse (b).*

by Hecht-Nielsen (1990), Hertz, Krogh, and Palmer (1991), and Kosko (1992). The back-propagation learning algorithm can be found in many places, having appeared originally in Rumelhart, Hinton, and Williams (1986). For the analog-computation and VLSI aspects of neural networks, the reference is Mead (1989). Various survey articles on neural networks exist, as the ones by Lee (1989), Palmer (1989), and Abu-Mostafa and Schweizer (1990). Jiang (1989) gives an update and more references on the firing squad problem. The chip firing problem was introduced by Spencer

(1986).

The taxonomy given in Section 1.2 for automaton networks is from Barbosa (1991).

2

Background

Some of the background needed throughout the book is presented in this chapter. Section 2.1 is devoted to a review of undirected and directed graphs, while Section 2.2 reviews the main intractability results within complexity theory. Probabilities and stochastic processes are discussed in Section 2.3. The pertaining bibliography is given in Section 2.4.

This chapter is by no means exhaustive on the topics it covers, nor does it cover all the background necessary to read the book. The topics selected for discussion in Sections 2.1 through 2.3 constitute material used throughout most of the chapters. Various other topics, of primary interest to one or a few chapters, are discussed when they are needed, often not as deeply as needed. Additional references are, however, always provided in the section on bibliography in the chapter.

2.1. GRAPHS

Our main interest in this section is to discuss basic concepts related to directed graphs. Before we do that, however, we introduce some additional concepts related to undirected graphs to complement the ones already seen in Chapter 1. The *degree* of a node in an undirected graph $H = (N_H, E_H)$ of node set N_H and edge set E_H is the number of neighbors the node has in H. A *path* in H is a sequence of nodes such that every node in it is a neighbor of the node that precedes it in the sequence. This path is said to be *between* the first and last nodes in the sequence. If these two nodes are the same node, then we have a *cycle*. If in H there exists a path between every two nodes, then H is a *connected* graph, otherwise each of the graphs formed by the subsets of N_H whose members are such that a path exists between every two of them is called a *connected component* of H. If H has no cycles, then it is called a *tree* if connected or a *forest* otherwise (in this case, each of its connected components is a tree). The *shortest path* between two nodes is the path that has fewest edges between those nodes. The *diameter* of H is the number of edges in the shortest path that has most edges. A cycle that does not go through any node more than once is a *simple cycle*, and is called a *Hamiltonian cycle* if it contains every node (H is in this case said to be Hamiltonian as well). A cycle that goes through all the edges in E_H exactly once is an *Eulerian cycle*, and exists if and only if in H every node has an even number of neighbors (in this case, H is said to be Eulerian itself). If in H multiple edges are allowed involving the same two nodes, then H is called a *multigraph*.

An undirected graph H is said to be *completely connected* if an edge exists connecting every two nodes in N_H. If the graph formed by taking nodes in a subset of N_H is completely connected, then that graph is said to be a *clique*. If a subset of E_H forms a tree in which all nodes in N_H participate, then the graph formed by that subset of edges on N_H is a *spanning tree* (a *spanning forest* is defined likewise). A *matching* is a subset of E_H whose edges share no nodes. If the inclusion of other edges in a matching can only be done if edges are allowed to share nodes, then the matching is *maximal*. A matching is *complete* if every node in N_H participates in one of its edges. A *node cover* is a subset of N_H whose nodes participate in every edge in E_H (it may be instructive to check that the complement of a node cover with respect to N_H is an independent set, introduced in Chapter 1). The *lexicographic product* of two undirected graphs H_1 and H_2, denoted by $H_1[H_2]$, is obtained by replacing every node of H_1 with an instance of H_2, and then interconnecting every node in the instance of H_2 used to replace a node in H_1 to every node in the instance of H_2 used to replace another node in H_1 if the two nodes in H_1 are connected by an edge. For $k, l \geq 1$, a *k-color, l-tuple coloring* of N_H (called a *node multicoloring* if $l > 1$) is an assignment of l different colors to each node in N_H such that no two neighbors receive a same color and furthermore k different colors are used overall. The same applies to multicolorings of E_H, in which case edges sharing a node are required to share no colors (if $l > 1$, an *edge multicoloring* is obtained).

A graph $H = (N_H, E_H)$ of node set N_H and edge set E_H is a *directed* graph if E_H is a set of ordered pairs from N_H^2. If E_H is symmetric, then H can also be regarded as an undirected graph by taking each symmetric pair of directed edges as a single undirected edge. In the general case, the *underlying undirected graph* of a directed graph H has the same node set as H and an edge set obtained from E_H by taking each ordered pair in it as an unordered pair and then eliminating duplicates so that the resulting undirected graph is not a multigraph. An edge (t_1, t_2) in a directed graph H is pictorially represented by an arrow drawn from t_1 to t_2. This edge is said to be *outgoing* from t_1 and *incoming* to t_2, or to be *directed from t_1 to t_2*. A *directed path* in H is a sequence of edges $(t_1, t_2), (t_2, t_3), \dots, (t_{k-1}, t_k)$ for some $k \geq 2$, and is said to be *outgoing* from t_1 (it is also said to be *from t_1 to t_k*). A *directed cycle* is a directed path in which $t_1 = t_k$. The graph H is an *acyclic directed graph* if it contains no directed cycles. If, on the other hand, every two nodes are on a same directed cycle, then H is said to be *strongly connected*, in which case a directed path exists from every node to every other node.

Alternatively, the edge set E_H may be thought of as the association of the edge set of H's underlying undirected graph, denoted by E_U, with an *orientation* ω, which is a function of the form

$$\omega : E_U \to N_H$$

such that $\omega(t_1, t_2) = t_2$ if and only if $(t_1, t_2) \in E_H$. An orientation ω is said to be an *acyclic orientation* if H is acyclic. Viewing directed graphs in this way is particularly useful in situations in which several directed graphs sharing the same underlying undirected graph must be considered concomitantly.

If H is an acyclic directed graph, then at least one of its nodes is such that no edge is outgoing from it; nodes with this property are called *sinks*. The set of sinks according to an acyclic orientation ω is denoted by $Sinks(\omega)$. Similarly, at least one of H's nodes is such that no edge is incoming to it, and is then called a *source*. The *sink decomposition* of an acyclic directed graph H is a partition of N_H into the λ subsets $S_0, \dots, S_{\lambda-1}$ for $1 \leq \lambda \leq |N_H|$, defined recursively as follows. S_0 is the set of sinks in H; for $0 < k \leq \lambda - 1$, S_k is the set of sinks in the directed subgraph of H formed by the subset of nodes $N_H - S_0 - \cdots - S_{k-1}$. Clearly, the sink decomposition of H has the property that, for $1 \leq k \leq \lambda - 1$, every node $t_1 \in S_k$ is such that at least one edge (t_1, t_2) exists with $t_2 \in S_{k-1}$. As a consequence, the number of edges in a longest directed path outgoing from t_1 is k (Figure 2.1).

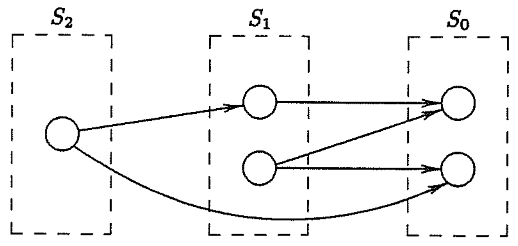


Figure 2.1. *The sink decomposition of an acyclic directed graph is obtained by repeatedly removing all the sinks from the graph, and grouping them in the sets S_0, S_1, etc. As sinks are removed to form one of these sets, new sinks appear, and the process is repeated until all nodes are exhausted.*

2.2. COMPLEXITY

For some $k \geq 1$, consider the k variables $y_1, \dots, y_k$, each taking values from a common finite domain D. Consider also the set $\mathcal{D} \subseteq D^k$, called the set of the *feasible points* in D^k. Given a function

$$\phi : D^k \to \mathbf{R},$$

a typical *combinatorial optimization problem*, or simply *optimization problem* in the context of this book, can be expressed as

$$\begin{array}{c}\text{minimize } \phi(y)\\ \text{subject to } y \in \mathcal{D},\end{array}$$

where y is a point in D^k whose components are the values of $y_1, \ldots, y_k$.

Depending on the form of ϕ and on the structure of $\mathcal{D}$, the degree of difficulty of finding a solution to this optimization problem can vary considerably, and is expressed as the *complexity* of the proposed algorithm. In the context of sequential computations, an important component of this complexity of an algorithm refers to the time for the algorithm to be run to completion on an abstract machine model (a *Turing machine*). Other measures exist, chiefly when other types of computation, as distributed parallel computations, are considered.

Complexity measures are in general expressed as asymptotic, worst-case functions of the size of the input to the problem, which we denote generically by s. For such, the following notation is quite useful. A function $g(s)$ is said to be $O(h(s))$ if two positive constants c and s_0 exist such that, for all $s \geq s_0$,

$$g(s) \leq ch(s).$$

Most of the study of the complexity of algorithms has concentrated on the so-called *decision problems*. The decision-problem form of the optimization problem introduced earlier in this section is the following. Given $\Phi \in \mathbf{R}$, is there a point $y \in \mathcal{D}$ such that $\phi(y) \leq \Phi$? If there exists a sequential algorithm to answer this question whose time complexity is expressed by a polynomial in s (e.g., $O(s^2)$), then the algorithm is agreed to be "efficient," mostly because problems for which no efficient algorithm has been found for arbitrary values of s can so far only be solved by techniques of exponential (i.e., $O(2^s)$) time complexity.

Decision problems that can be solved in polynomial time constitute a class P (for *Polynomial*). Problems in this class are also members of another class, called *NP* (for *Nondeterministic Polynomial*), defined as follows. A decision problem is said to belong to *NP* if, given a point $y \in D^k$ such that $y \in \mathcal{D}$ and $\phi(y) \leq \Phi$, there exists a polynomial-time algorithm to verify that indeed $y \in \mathcal{D}$ and $\phi(y) \leq \Phi$. This class of decision problems can be thought of as capturing the "easy part" of many problems known in practice to be of difficult solution, i.e., problems for which finding y has not been shown to be possible in polynomial time. Just as the class P can be regarded as the "easy" portion of *NP*, its hardest portion, composed of the so-called *NP*-complete problems, has also been extensively characterized.

A decision problem Π is said to be *NP-complete* if it belongs to *NP* and furthermore every other problem Π' in *NP* can be indirectly solved by solving Π, as long as the time to transform an instance of Π' into an instance of Π is polynomial in s. Clearly, by this definition the *NP*-complete problems are the hardest problems within *NP*. In addition, if any *NP*-complete problem is found to be solvable in polynomial time, then $P = NP$. Whether such an *NP*-complete problem exists is one of the most celebrated open questions in computer science.

Returning to our original optimization problem, we note that, once a solution $y^* \in \mathcal{D}$ is found to it, this same solution can be used to reply to the question posed by the problem's decision-form variant, affirmatively if $\phi(y^*) \leq \Phi$, negatively otherwise. Then the optimization problem is seen to satisfy the definition of *NP*-complete problems if its decision-problem variant is in *NP*, except for its own membership in *NP*. Problems with this characteristic are called *NP-hard* problems, because, by definition, they are at least as hard as the problems in *NP* (Figure 2.2).

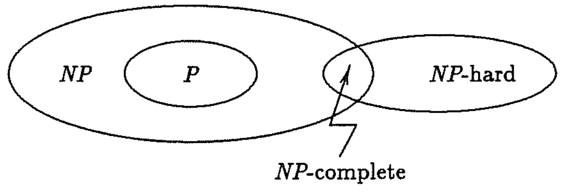


Figure 2.2. *The class NP of decision problems has two important subclasses. One of them is the class P, comprising problems that admit a polynomial-time solution. The other class is the class of the NP-complete problems, constituting by definition the hardest problems within NP. This class of NP-complete problems can be regarded as being the intersection of the class NP with the class of the NP-hard problems.*

2.3. PROBABILITIES AND STOCHASTIC PROCESSES

Consider a set S, and the set 2^S of its $2^{|S|}$ subsets. Within the context of probabilistic situations, the set S is a *sample space*, and 2^S is normally referred to as a set of *events*, but we shall refrain from utilizing this latter denomination in other chapters to avoid confusion with the homonymous concept discussed elsewhere in the book. A *probability distribution*, or simply *distribution* P on 2^S is a real function on 2^S for which the following three properties hold.

(i) $0 \leq P(a) \leq 1$ for all $a \in 2^S$;

(ii) $P(S) = 1$;

(iii) If $a \cap b = \emptyset$ for $a, b \in 2^S$, then $P(a \cup b) = P(a) + P(b)$.

From (ii) and (iii), it follows that

$$\sum_{s_1 \in S} P(s_1) = 1,$$

where we have taken $P(s_1)$ as a simpler notation for $P(\{s_1\})$.

We make a distinction between the notation $P(a)$, which can be interpreted as meaning "the probability of event a according to P," and the notation $\Pr(a)$, which we shall take to mean simply "the probability of event a," regardless of the distribution. This is useful in situations in which the distribution is unknown or needs to be left unspecified for some reason.

The notation $a \mid b$ is to be understood as "a given b" or "a conditioned upon b," and is used to define the *conditional probability* of event a given event b (or conditioned upon event b) when $P(b) > 0$ as

$$P(a \mid b) = \frac{P(a \cap b)}{P(b)}.$$

For $k \geq 2$, events $a_1, \ldots, a_k$ are said to be *independent* of one another if and only if

$$P(a_{i_1} \cap a_{i_2}) = P(a_{i_1})P(a_{i_2}),$$

and

$$P(a_{i_1} \cap a_{i_2} \cap a_{i_3}) = P(a_{i_1})P(a_{i_2})P(a_{i_3}),$$

and so on, all the way through

$$P(a_1 \cap \cdots \cap a_k) = P(a_1) \cdots P(a_k),$$

for all $1 \leq i_1 < \cdots < i_{k-1} \leq k$.

If the events $a_1, \ldots, a_k$ are such that

$$a_1 \cup \cdots \cup a_k = S$$

and

$$a_{i_1} \cap a_{i_2} = \emptyset$$

for all i_1, i_2 such that $1 \leq i_1 < i_2 \leq k$, then the *total probability formula*,

$$\begin{aligned} P(b) &= \sum_{\ell=1}^{k} P(a_\ell \cap b) \\ &= \sum_{\ell=1}^{k} P(b \mid a_\ell)P(a_\ell), \end{aligned}$$

holds for every event b. This is also the case for

$$P(a_l \mid b) = \frac{P(b \mid a_l)P(a_l)}{\sum_{\ell=1}^{k} P(b \mid a_\ell)P(a_\ell)},$$

known as *Bayes's formula*, for all $1 \leq l \leq k$.

Given a finite set D, a *discrete random variable* y, or simply *random variable* in the context of this book, is a mapping

$$y : S \to D.$$

For a distribution P on 2^S and some $d_1 \in D$, we let $y = d_1$ stand for the event a such that $y(a) = d_1$, and write

$$P(d_1) = P(y = d_1).$$

Clearly,

$$\sum_{d_1 \in D} P(d_1) = 1.$$

The *expected value* (or *average*) of y given P is

$$\sum_{d_1 \in D} d_1 P(d_1).$$

Random variables are often considered in groups, and then a few adjustments have to be made to the concepts seen thus far. If $y_1, \ldots, y_k$ are random variables for some $k \geq 1$ and $d_1, \ldots, d_k \in D$, then $y_1 = d_1, \ldots, y_k = d_k$ stands for the event $a_1 \cap \cdots \cap a_k$ such that $a_1, \ldots, a_k$ are events for which $y_\ell(a_\ell) = d_\ell$, where $1 \leq \ell \leq k$. We then write

$$P(d_1, \ldots, d_k) = P(y_1 = d_1, \ldots, y_k = d_k).$$

The distribution P is in this case referred to as a *joint distribution* over 2^S, and sometimes, more loosely, as a distribution on D^k. Clearly,

$$\sum_{(d_1, \ldots, d_k) \in D^k} P(d_1, \ldots, d_k) = 1.$$

A family of random variables indexed by an integer parameter,

$$y(0), y(1), y(2), \ldots,$$

is a *discrete-parameter stochastic process.* Each of these random variables is a mapping with common range, i.e.,

$$y(\tau) : S \to D$$

for $\tau \geq 0$. Often the parameter τ has a connotation as time, and then the discrete-parameter stochastic process is referred to as a *discrete-time stochastic process.* The *state space* of a stochastic process is the range of its random variables, in our case D. As the state space is in our case discrete, we generally omit the additional qualification as a "discrete-state" stochastic process. A central question in the analysis of discrete-parameter stochastic processes is to specify the joint distribution of the random variables $y(0), \ldots, y(\tau)$ for $\tau \geq 0$.

In the discrete-parameter case, *Markov chains* are discrete-state stochastic processes whose joint distribution P satisfies

$$P\big(y(\tau) = d_\tau \mid y(0) = y_0, \ldots, y(\tau - 1) = d_{\tau-1}\big) = P\big(y(\tau) = d_\tau \mid y(\tau - 1) = d_{\tau-1}\big)$$

for all $\tau > 0$. This is the *one-step transition probability*, or simply *transition probability*, denoted by

$$T(\tau, d_1, d_1') = P\big(y(\tau) = d_1 \mid y(\tau - 1) = d_1'\big)$$

for all $d_1, d_1' \in D$, and causes the chain to be called *homogeneous* if it does not depend on τ (i.e., is the same for all $\tau > 0$). In the homogeneous case, we simplify the notation to $T(d_1, d_1')$.

Because our state space is finite, every homogeneous Markov chain with strictly positive transition probabilities admits a *stationary distribution* P (stochastic processes with this characteristic are called *stationary processes*), which is itself strictly positive at all states and given as the unique solution to the linear system

$$P(d_1) = \sum_{d_1' \in D} P(d_1')T(d_1, d_1')$$

for all $d_1 \in D$, subject to

$$\sum_{d_1 \in D} P(d_1) = 1.$$

If, in addition,

$$P(d_1')T(d_1, d_1') = P(d_1)T(d_1', d_1)$$

for all $d_1, d_1' \in D$, then the Markov chain is said to be *reversible* (this may also occur for chains that are not homogeneous but do nonetheless admit a stationary distribution).

2.4. BIBLIOGRAPHIC NOTES

The material on graphs presented in Section 2.1 can be complemented by the books on graph theory by Harary (1969), Berge (1976), Wilson (1979), and Bondy and Murty (1976). Graph algorithms are discussed by Even (1979). Node multicolorings are treated by Stahl (1976), and edge multicolorings by Fiorini and Wilson (1977) and Stahl (1979). The concept of a sink decomposition of an acyclic directed graph can be found in Berge (1976) and Barbosa and Gafni (1987, 1989b).

Additional information on the complexity of algorithms (Section 2.2) can be found in the pioneering paper by Cook (1971), where the first *NP*-completeness result was shown, in Karp (1972), where various of the most important *NP*-completeness results are given, and in Garey and Johnson (1979), a comprehensive

book on the subject. Chapters dealing with this theme are given by Papadimitriou and Steiglitz (1982) and by Cormen, Leiserson, and Rivest (1990).

The discussion in Section 2.3 on probability theory can be enlarged by the books by Feller (1968), Rozanov (1969), Feller (1971), and Karlin and Taylor (1975, 1981). A review chapter on the subject is given by Kleinrock (1975), while Trivedi (1982) discusses various applications.

Part 2

Fundamentals of distributed parallel computation

Chapters in this part are devoted to a discussion, at an abstract level, of issues related to the design of distributed parallel algorithms in general and to the design of distributed parallel simulators of automaton networks belonging to the two classes treated in this book.

Chapters 3 and 4 constitute Part 2. Chapter 3 contains a discussion of distributed algorithms in general, along with some techniques that have acquired a status as fundamental for the design of those algorithms. Chapter 4 continues with the discussion of distributed algorithm techniques, now aiming at the development of distributed parallel simulators for FC and PC automaton networks.

3

Models of distributed parallel computation

This chapter contains a discussion at an abstract level of various issues pertaining to the design and analysis of distributed algorithms. Basic topics are treated in Section 3.1, as models of distributed parallel computation, complexity measures, and notation. Sections 3.2 and 3.3 are devoted to two of the most important techniques used in the design of distributed algorithms, namely the recording of global states and the detection of the computation's termination. Bibliographic notes appear in Section 3.4.

3.1. DISTRIBUTED ALGORITHMS

3.1.1. The nature of distributed algorithms

Parallel processing systems are characterized fundamentally by the existence of multiple *agents* that cooperate in the execution of a task. Depending on the level of abstraction at which this parallel processing system is described, the notion of an agent may embody several different entities, and therefore imply various patterns of behavior and characteristics. For example, if the system under consideration is a parallel machine, then an agent is naturally identified with one of the processors in that machine. At a higher level of abstraction, the system might be the set of processes that constitute a parallel program, and in this case an agent would be identified with one of the processes. At yet another level of abstraction (a lower one), the system might be the hardware of a single processor, and then an agent would be one of the functional units that concur for the proper functioning of the processor.

From a strictly conceptual point of view, it does not matter much whether the parallelism is present at the hardware level or at the topmost software level. In fact, parallel processing agents can be easily identified in as seemingly disparate settings as operating systems and computer networks, for example. The essence here is that nearly every fundamental concept related to the interaction of those parallel processing agents has its counterpart in all of these domains. This includes, for example, the celebrated notions of concurrency, deadlock, starvation, and so on, however differently they may manifest in each situation.

Despite this apparent ubiquity of parallel processing system characteristics, this denomination is usually reserved for systems that do not spread geographically too widely, as do computer networks, but rather are confined within a single machine. Even in such a restricted context, the presence of parallelism is pervasive, and appears at many different levels. Within the scope of this book, however, the primary interest is to identify the characteristics that allow such a parallel processing system to be called a *distributed system*.

Usually, we say that a parallel processing system is a distributed system if its agents must rely on message passing as the only means for communicating with one another. Note that this same characterization holds within the broader context we discussed earlier, as the mechanisms of message passing seem to be just as fundamental in computer networks and in concurrent programs whose processes communicate via the exchange of messages even when residing in a single processor. Within the more restricted context, the use of message passing as the sole means of communication is intimately related to the architecture of the parallel processing machine at hand, inasmuch as such a strict constraint can only hold for distributed-memory machines.

Researchers in the field of distributed computing still lack a consensus regarding an acceptable definition of *distributed algorithms*. In this book, however, we shall view a distributed algorithm as an algorithm designed to run on a distributed system understood as a set of agents that communicate by message passing. An agent in such a system has a *reactive*, or *message-driven*, behavior, in the sense that it performs actions upon the reception of messages only. An exception to this general pattern of behavior must be allowed at the onset of the distributed computation, but this depends on the particular model of distributed computation at hand, and will be discussed in detail in Section 3.1.2.

This view of a distributed system and of a distributed algorithm seems only natural given the class of computations we are interested in, namely the parallel simulation of automaton networks, as introduced in Chapter 1. In such a network, whose structure is represented by the undirected graph $G = (N, E)$, a node in N can be regarded as a parallel processing agent, and then the use of message passing for inter-agent communication, as well as the message-driven nature of each agent, becomes obvious. Loosely speaking, a node can only perform "computation" upon receiving "messages" from its neighbors in G, where these two terms are to be understood, respectively, as the updating of the node's state and the reception of its neighbors' updated states.

Throughout most of this book, the parallel processing agents will be referred to as *processors*, meaning that in most of our discussion it will be assumed that there exists a processor responsible for the simulation of each node in N. These processors will, in addition, be assumed to be able to communicate with one another

by means of bidirectional communication channels laid between pairs of processors in accordance with the edges in E. Viewing our parallel processing system in such a way is of course an oversimplification, as it allows us to disregard the very important issues of process allocation and scheduling, among others. This approach is, nevertheless, unharmful from the standpoint of distributed algorithm design. In Appendix A, we discuss how such unreal assumptions can be relaxed in such a way that the functionality of the algorithms is uncompromised.

3.1.2. Models of distributed parallel computation

Our models of distributed parallel computation are based on the graph $G = (N, E)$ introduced in Chapter 1. In G, N is a set of nodes and E is a set of undirected edges. Recalling that $n = |N|$, for $1 \le i \le n$ a processor p_i is associated with node n_i. Processors associated with nodes that are connected by an edge in E, i.e., neighbor nodes, are said to be *neighbors* as well, and may communicate with each other through a bidirectional communication channel that interconnects them directly. This characterizes what is called a *point-to-point distributed system*.

Processor p_i is the sole responsible for updating n_i's state during the automaton network simulation. In this chapter and in part of Chapter 4, however, the material we discuss is applicable regardless of p_i's particular function, so p_i shall be treated as a processor in a generic point-to-point distributed system.

In contrast with the case of sequential computations, for which the RAM (Random Access Machine) model is used almost exclusively, a great number of models exist for parallel computations. The primary reason for this diversity is the great variety of architectures for parallel processing, which makes it very difficult to adopt a model that represents all architectures (or even most of them) adequately. As opposed to the case of sequential machines, any such model would necessarily be distant from many real-world machines enough to be practically useless in those cases.

If for shared-memory machines the PRAM (Parallel Random Access Machine) model seems to be largely favored, there is nothing close to consensus when it comes to distributed-memory machines. Of crucial importance, the appropriate choice of a model is especially relevant for complexity-related issues, as we discuss in Section 3.1.4. Nevertheless, some of the models' characteristics are general enough to be discussed relatively safely across the borders from one model to others. One such aspect is that of *timing*, or, put differently, the *synchronism* (or *asynchronism*) of a model.

There is a wide spectrum of possibilities regarding the timing of a model of distributed computation. At one extreme there is the *fully synchronous* (*synchronous*, for short) *model*, and at the other the *fully asynchronous* (*asynchronous*, for short) *model*.

The synchronous model is characterized by the following two properties.

- A global time basis that drives all processors.
- An upper bound on the time it takes a message sent between neighbors to be delivered.

These two properties are related to each other by the requirement that the delay for message delivery between neighbors be no larger than the duration of an interval of the global clock.

The counterparts of these two properties for the asynchronous model are the following.

- Each processor is driven by its own, local, independent time basis.
- A message sent between neighbors is subject to a delay to be delivered that is unpredictable, although assumed to be finite.

These two sets of characteristics constitute the main differences between the synchronous and the asynchronous models, and have considerable impact on the way of measuring the computational complexity of algorithms. The two models do, however, have characteristics in common. One of these characteristics is the usual assumption that the computation performed locally by a processor is fast enough (when compared to the time for communication) to be thought of as occurring in zero time. This assumption is usually justified by the fact that, in message-passing systems, the dominant cost is that of communication. While this is true in the case of the algorithms treated in this book, it is not so in general.

Another characteristic that the synchronous and asynchronous models have in common concerns the *capacity* of each of the bidirectional communication channels. This notion is explained in more detail in Appendix A, and has nothing to do with the electromagnetic capacity of the physical channel, expressed in bits per second. Instead, it indicates the number of messages that the channel can hold (buffer) before either a message must be received by the destination processor or the origin processor must be blocked upon attempting to send another message on the same channel. In both the synchronous and the asynchronous models, this capacity is infinite for all channels, so processors never get blocked upon attempting a message transmission. This assumption makes the design of distributed algorithms much more comfortable than it would be otherwise, but is nevertheless unreal and must be dealt with somewhere along the process of obtaining a real distributed parallel program.

Distributed algorithms designed under the assumption of the synchronous model function along the following generic lines. At the beginning of each pulse of the global clock, a processor computes locally on the messages it received in the previous pulse, and then sends messages out, if at all. An exception to this general pattern occurs for the first pulse, at which at least one processor must send messages out "spontaneously," i.e., not in response to a message it received previously. One interesting consequence of the assumption that local computation takes no time is that, in the synchronous model, every message sent by a processor to one of its neighbors during a certain clock pulse necessarily reaches its destination before the end of that pulse. This is so because, by the assumption, such messages must have been sent at the beginning of the clock pulse, and then must, by the model's characteristics, suffer a delay no larger than the duration of the pulse.

Under the assumption of the asynchronous model, on the other hand, a processor computes locally upon receiving a message, and may then send messages out

as a consequence. Just as in the synchronous case, the algorithm is initiated by at least one processor, which sends messages out spontaneously. The only difference is that, in the asynchronous case, this spontaneous activity has no definite point in time at which to happen (time is in this case a local concept, and bears no relation to its counterparts in other processors).

The two models we discussed in this section are highly useful, each one with its particular appeal. The synchronous model does in general make the design of a distributed algorithm significantly simpler, because in this model the fact that a message is not received from a certain neighbor at the end of a certain clock pulse has a meaning that can be exploited in designing the algorithm — it means that no message was sent by that neighbor during that pulse. The same property can never be expected to hold in the asynchronous model, in which messages suffer unpredictable delays.

The asynchronous model, on the other hand, is much more realistic from the standpoint of most distributed-memory parallel machines, and therefore it is under the assumptions of this model that the ultimate algorithm must be designed. In Chapter 4, we shall see that this apparent conflict between ease of design and reality can be solved rather easily by the use of the so-called synchronizers.

3.1.3. Some formalism

For the remainder of Section 3.1 and for Section 3.2, it shall be useful to introduce some formalism into the notions of a distributed system, its model, and the algorithms designed for it. While our discussion in all of Section 3.1 is applicable to both the synchronous and the asynchronous models of distributed computation, it is under the asynchronous model that this formalism plays a more important role. We shall, therefore, utilize the latter model throughout, with occasional remarks on how the concepts would appear under the synchronous model.

The formalism we utilize is based on the concept of an *event* as the fundamental unit of a distributed computation. A distributed computation is then in this formalism simply a set of events, which we denote by Ξ. An event $\xi \in \Xi$ is the 6-tuple

$$\xi = (p_i, t, \varphi, \sigma, \sigma', \Phi),$$

where

- p_i is the processor where the event occurs, for some $1 \leq i \leq n$;
- t is the time, as given by p_i's local clock, at which the event occurs;
- φ is the message, if any, that triggered the event upon its reception by p_i;
- σ is the state of p_i prior to the occurrence of the event;
- σ' is the state of p_i after the occurrence of the event;
- Φ is the set of messages, if any, sent by p_i as a consequence of the occurrence of the event.

This definition of an event is based on the premise that the behavior of each processor can be described as that of a state machine, which seems to be general enough. The computation Ξ then causes every processor to have its state evolve as the events occur. We let Σ_i denote the sequence of states p_i goes through as Ξ goes on. The first member of Σ_i is p_i's *initial state*. The last member of Σ_i (which may not exist if Ξ is not finite) is p_i's *final state*.

This definition of an event is also general enough to encompass both the reactive character we wish to embed in our distributed computations and to allow the description of "internal events," i.e., events that happen without any apparent external cause. This is why, in the definition, we have allowed the input message φ associated with an event ξ to be absent sometimes. As we shall see shortly, these internal events are not to be confused with the spontaneous events associated with the beginning of a computation at some processors. Spontaneous events must be allowed to happen with no cause at all, while these internal events follow the normal causality flow inside each processor, and can ultimately be seen to have been caused either by the reception of a message or by a spontaneous event.

Most of this definition would still be valid if we were using a synchronous model of distributed computation. Two aspects, however, might be reconsidered. First, the time t at which an event occurs would in the synchronous model be restricted to the values at which the global clock "ticks," for example the nonnegative integers. Secondly, as in the synchronous model the concept of "triggering an event" is related to the reception of a nonzero number of messages (all events happen at the beginning of clock pulses and compute on whichever messages were received at the previous pulse), we might as well allow for the reception of multiple messages (instead of the single message φ) in association with an event.

Events in Ξ are strongly interrelated, as messages sent out in connection with one event are received in connection with others. While this dependency is already grasped by the definition of an event, it is useful to elaborate a little more on the issue. Let us define a binary relation, denoted by $\prec$, on the set of events Ξ as follows. If ξ_1 and ξ_2 are events, then $\xi_1 \prec \xi_2$ if and only if one of the following two conditions is satisfied.

(i) Both ξ_1 and ξ_2 occur at the same processor, respectively at (local) times t_1 and t_2 such that $t_1 < t_2$. In addition, no other event occurs at the same processor at a time t such that $t_1 < t < t_2$.

(ii) ξ_1 and ξ_2 occur at neighbor processors, and a message φ exists that is sent in connection with ξ_1 and received in connection with ξ_2.

It follows from conditions (i) and (ii) that $\prec$ is an acyclic relation. Condition (i) expresses our intuitive understanding of the causality that exists among events that happen at a same processor, while condition (ii) gives the basic cause-effect relationship that exists between neighbor processors.

One interesting way to view the relation $\prec$ defined by these two conditions is to consider the acyclic directed graph $H = (\Xi, \prec)$. The node set of H is the set of events Ξ, and its set of edges is given by the pairs of events in $\prec$. The graph H is a *precedence graph*, and can be pictorially represented by displaying the nodes

(events) associated with a same processor along a horizontal line, in the order given by $\prec$. In this representation, horizontal edges correspond to pairs of events that fall into the category of condition (i), and all others are in the category of condition (ii). Equivalently, horizontal edges can be viewed as representing the states of processors (only initial and final states are not represented), and edges between the horizontal lines of neighbor processors represent messages sent between those processors (Figure 3.1). Viewing the computation Ξ with the aid of this graph will greatly enhance our understanding of some important concepts to be discussed in Section 3.2.

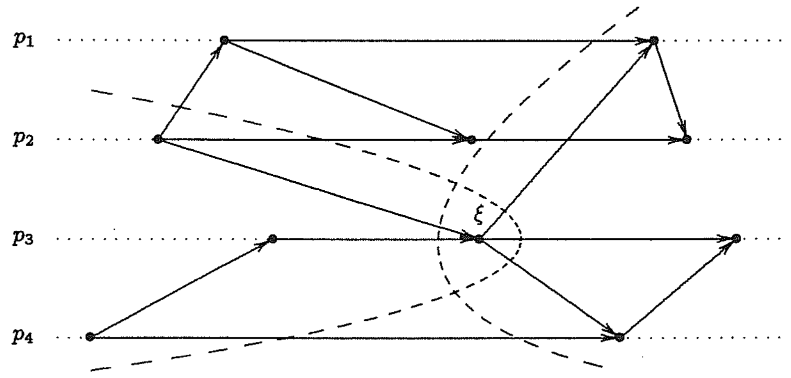


Figure 3.1. *A precedence graph has the events in Ξ for nodes and the ordered pairs in the partial order $\prec$ for directed edges. It is visually appealing to draw precedence graphs so that events happening at a same processor are placed on a horizontal line and positioned on this line, from left to right, in increasing order of the local times at which they happen. In this figure, the "conically"-shaped regions delimited by dashed lines around event ξ happening at processor p_3 represent $\{\xi\} \cup Past(\xi)$ (the one on the left) and $\{\xi\} \cup Future(\xi)$ (the one on the right).*

The transitive closure of $\prec$, denoted by $\prec^+$, is irreflexive and transitive, and therefore establishes a *partial order* on the set of events Ξ. Two events ξ_1 and ξ_2 unrelated by $\prec^+$, i.e., such that

$$(\xi_1, \xi_2) \in \Xi \times \Xi - \prec^+$$

and

$$(\xi_2, \xi_1) \in \Xi \times \Xi - \prec^+,$$

are said to be *concurrent events*.

To finalize this section, we use the relation $\prec^+$ to define an event's *past* and *future*. For an event ξ, we let

$$Past(\xi) = \{\xi' \mid \xi' \in \Xi \text{ and } \xi' \prec^+ \xi\}$$

and

$$Future(\xi) = \{\xi' \mid \xi' \in \Xi \text{ and } \xi \prec^+ \xi'\}.$$

These two sets can be easily seen to induce "conical" regions emanating from ξ in the precedence graph H (Figure 3.1).

3.1.4. Complexity measures

The complexity of a sequential computation is evaluated by computing the overall number of "relevant" operations, i.e., operations whose computational cost is agreed to have a significant impact on the computation's time for completion. Parallel computations have their complexity evaluated in much the same manner, although the choice of relevant operations is no longer simple or even consensual, but depends, as expected, very intimately on the model of computation adopted.

For a number of models of parallel computation (including models of distributed parallel computation), the complexity is evaluated by counting the number of processors employed and the time required for the computation to terminate. As these models tend to be all synchronous, in the sense discussed in Section 3.1.2, the concept of time is well defined and refers to the number of clock pulses (or, more coarsely, the number of "steps") elapsed.

In our case, however, the number of processors as a complexity measure is immaterial, as we have assumed it fixed and equal to n, the number of nodes in G. In addition, as we have chosen to adopt a model of parallel computation in which the cost of message passing dominates all others, there has to be a measure that takes into account the communication cost that an algorithm incurs.

The traditional way of measuring complexity in distributed systems with the characteristics of our system is to compute the overall number of messages exchanged (*communication complexity*) and the overall time elapsed (*time complexity*). Given an algorithm, say *Alg*, we shall sometimes denote its communication complexity by *Comm*(*Alg*), and its time complexity by *Time*(*Alg*). These are, as usual, expressed as asymptotic, worst-case measures over all possible executions of *Alg*.

Just as we remarked previously, a little more discussion is needed to clarify how to measure time. Under the assumption of a synchronous model of computation, time is measured by simply counting the number of clock pulses elapsed. For the asynchronous model, on the other hand, no such global reference exists, and we must resort to the formalism developed in Section 3.1.3. Within this formalism, the time complexity associated with a computation (set of events) Ξ is the number of message-related edges in the directed path in the precedence graph H for which this number is greatest. This is, in other words, a way of measuring the length of

the longest "causal chain" in the computation, taking into account the fact that the cost of internal processing is negligible.

Let us consider, for the sake of example, the following simple distributed algorithm problem, known as *Propagation of information with feedback — Pif*. Informally, a processor, say p_1, holds a piece of information, *inf*, which has to be disseminated among all other processors in the system. In addition, p_1 must somehow be able to detect that all processors have already received *inf*.

The synchronous algorithm for Pif, let us call it Algorithm *S_Pif*, is very simple. It begins with p_1 sending *inf* to all of its neighbors. These, at the beginning of the next clock pulse, send *inf* to their neighbors, which do the same when a new pulse begins, unless they have already done so in some previous pulse. The algorithm then proceeds in a similar fashion until no processor sends any more messages. In order that p_1 detects the global termination of the computation, it suffices that it keep track of the evolution of time until $n-1$ or D (G's diameter) pulses have elapsed, depending on which of n or D can be assumed to be known by p_1. It should be immediate to understand that at this time all processors will surely have received their copies of *inf*. (The issue of distributed algorithm termination is discussed in a more global context in Section 3.3.)

The correctness of Algorithm *S_Pif* is immediate to be established. Similarly, its time complexity is $O(n)$ (or $O(D)$), while its communication complexity is $O(|E|)$, as every processor sends exactly one message to each of its neighbors, and therefore each edge in E carries exactly two messages, one in each direction.

The asynchronous algorithm for Pif, call it Algorithm *A_Pif*, is not so simple, as the evolution of local time has no global significance in the asynchronous model. In Algorithm *A_Pif*, processor p_i employs a variable $parent_i$, initially equal to **nil**. Initially, p_1 sends *inf* to all of its neighbors. Processor $p_i \neq p_1$, upon receiving *inf* for the first time, lets $parent_i$ contain the identification, say p_j, of the neighbor from which *inf* was received, and then sends *inf* to all of its neighbors, except $parent_i$. When a generic processor p_i has received *inf* from all of its neighbors, it sends *inf* to $parent_i$, unless $i = 1$, in which case $parent_i =$ **nil** and the reception of *inf* from all neighbors signifies that the algorithm has terminated globally, i.e., every processor has its copy of *inf*.

The correctness of Algorithm *A_Pif* is not as immediate as its synchronous counterpart's, and calls for a slightly more formal argument. Let us say that Algorithm *A_Pif terminates* at a processor p_i when p_i has received *inf* from all of its neighbors and, if $i \neq 1$, sent *inf* to $parent_i$.

Theorem 3.1. *Algorithm A_Pif terminates at all processors, and upon termination at p_1 every processor has received inf.*

Proof: The variables $parent_i$ induce a spanning tree on G, as clearly every processor p_i receives *inf* for the first time exactly once. This tree can be regarded as being rooted at n_1, and then its leaves correspond to those processors from which no other processor received *inf* for the first time (Figure 3.2). The proof proceeds by induction on the subtrees of this tree. Along the induction, the assertion to be shown is that Algorithm *A_Pif* terminates at all processors corresponding to nodes in the subtree, and that upon termination at the processor corresponding to the subtree's root every processor corresponding to a node in the subtree has received *inf*.

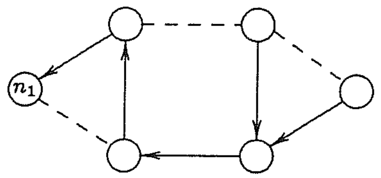


Figure 3.2. *Along the execution of Algorithm A_Pif, the variables* $parent_i$ *for each processor* p_i *are set so that a spanning tree is created on* G. *This spanning tree is rooted at* n_1, *and its leaves correspond to processors from which no other processor received inf for the first time. In this figure, a directed edge is drawn from* n_i *to* n_j *to indicate that* $parent_i = p_j$.

The basis of the induction is given by the subtrees rooted at the leaves, and then the assertion clearly holds, as no leaf n_i is such that $p_i = parent_j$ for some processor p_j. As the induction hypothesis, assume the assertion for the subtrees rooted at nodes corresponding to processors p_j such that $parent_j = p_1$. Then processor p_1 receives *inf* from all of its neighbors, and Algorithm *A_Pif* terminates at p_1, at which time *inf* has been received at all processors. ■

It follows from Theorem 3.1 that the communication complexity of Algorithm *A_Pif* is, as in the synchronous case, $O(|E|)$, as each edge in E is traversed by two messages, one in each direction. In order to evaluate Algorithm *A_Pif*'s time complexity, we argue as follows. The longest causal chain in this computation happens when the variables $parent_i$ have been set in such a way as to induce on G a spanning tree of height n rooted at n_1. The length of such a chain is $2(n-1)$, so the time complexity of Algorithm *A_Pif* is $O(n)$.

3.1.5. Notation

In this section, we introduce the notation to be employed throughout the book for the description of distributed algorithms. Each algorithm will be described by giving, for processor p_i, $1 \leq i \leq n$, a set of variables with the corresponding initial values, and a set of input/action pairs describing the behavior of p_i upon the reception of external inputs. For synchronous algorithms these external inputs are the value of the global clock, indicating the beginning of a new clock pulse, and the (possibly empty) set of messages received during the previous pulse. The clock

pulse will be denoted by the integer $c \geq 0$, and the set of input messages by MSG. For asynchronous algorithms the external inputs consist of the reception of a single message, denoted by msg. We assume that each action is executed in response to its input in an *atomic* manner, i.e., without interrupts, so we need not worry about the sharing of variables by the various actions as far as the variables' integrity is concerned.

In order to illustrate the usage of this notation for distributed algorithm description, let us employ it on Algorithms *S_Pif* and *A_Pif*, introduced in Section 3.1.4 for the synchronous and asynchronous propagation of a piece of information, *inf*, by p_1 with feedback. Recall that D stands for G's diameter and that $\mathcal{N}(n_i) \subseteq N$ is the set of n_i's neighbors in G.

In Algorithm *S_Pif*, processor $p_i \neq p_1$ maintains a variable $received_i$ to indicate whether *inf* has already been received.

```
Algorithm S_Pif:
  ▷ Variables:
        $received_i$ = false.
  ▷ Input:
        $c = 0$, $MSG = \emptyset$.
    Action at $p_1$:
        $received_i$ := true;
        Send inf to every $p_j$ such that $n_j \in \mathcal{N}(n_1)$;
        Wait until $D$ pulses have elapsed.
  ▷ Input:
        $c > 0$, $MSG = \{inf, \ldots, inf\}$.
    Action:
        if $MSG \neq \emptyset$ then
            if not $received_i$ then
                begin
                    $received_i$ := true;
                    Send inf to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
                end.
```

It should be noted that the first input/action pair in this description of Algorithm *S_Pif* refers solely to p_1, as indicated. As p_1 is the algorithm's initiator, such an input has no effect in the other processors.

In Algorithm *A_Pif*, processor p_i maintains, in addition to the variable $received_i$ of Algorithm *S_Pif*, a variable $parent_i$ to indicate the neighbor from which *inf* was first received.

```
Algorithm A_Pif:
  ▷ Variables:
        $received_i$ = false;
        $parent_i$ = nil.
  ▷ Input:
        $msg$ = nil.
    Action at $p_1$:
        $received_i := $ true;
        Send $inf$ to every $p_j$ such that $n_j \in \mathcal{N}(n_1)$.
  ▷ Input:
        $msg = inf$.
    Action:
        if not $received_i$ then
            begin
                $received_i :=$ true;
                $parent_i := origin_i(msg)$;
                Send $inf$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$ and $p_j \neq parent_i$
            end;
        if $seq_i(msg) = |\mathcal{N}(n_i)|$ then
            if $parent_i \neq$ nil then
                Send $inf$ to $parent_i$.
```

As in the case of Algorithm *S_Pif*, and for the same reason, the first input/action pair refers to p_1 only. In addition, in this description of Algorithm *A_Pif* we have employed some additional notation, which will be found elsewhere in the book as well. We have utilized $origin_i(msg)$ to indicate the neighbor of p_i from which msg was received, and $seq_i(msg)$ to indicate the number of messages already received by p_i during the execution of the algorithm, from its beginning up to, and including, the reception of msg.

As one last remark, it is important to note that, in the algorithms we just discussed, processors have "knowledge" of their identities. This assumption, which in most contexts is only natural to be made, is instrumental in the design of a variety of distributed algorithms, in contrast with "anonymous" systems, which impose severe impossibility conditions on the classes of computation that can be performed. Another implicit assumption has been that the processors under the synchronous model have knowledge of n (or D), the system size. While we could have done without this assumption, it is not unreasonable and has helped in illustrating one of the most important practical differences between the synchronous and asynchronous models, namely that simply "waiting" in the synchronous model can yield meaningful information to a processor. The absence of this assumption is also critical in that it limits somewhat the range of possible computations.

3.2. DISTRIBUTED SNAPSHOTS

3.2.1. Global states of a distributed computation

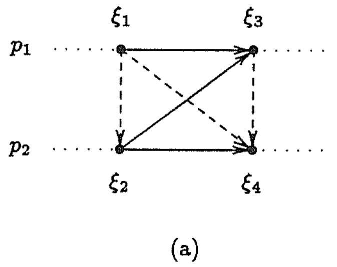
The concept of time in distributed systems is very elusive. While in the synchronous model the presence of the global clock makes it easy to deal with system-wide situations, the same is not true under the assumptions of the asynchronous model, in which only local time bases are provided. Take for example the Algorithms *S_Pif* and *A_Pif* discussed in Sections 3.1.4 and 3.1.5, respectively for a synchronous and an asynchronous model of distributed computation. Regarding Algorithm *S_Pif*, we can correctly state that "at time $n-1$ all processors have received *inf*," where *inf* is the piece of information being propagated. On the other hand, for Algorithm *A_Pif* such a statement is meaningless, and so is any attempt to express the fact that all processors have received *inf* in simple time-related terms.

One very helpful notion is that of a *consistent global state*, or simply *global state* (or yet *snapshot*). This notion is based on the formalism developed in Section 3.1.3, and, among other interesting properties, allows several global properties of distributed systems to be referred to properly in the asynchronous model. We will in this section provide two definitions of a global state. While these two definitions are equivalent to each other, each has its particular appeal, and is more suitable to a particular situation. Our two definitions are based on the weaker concept of a *system state*, which is simply a collection of n local states, one for each processor, and $2|E|$ channel states, one for each communication channel in each direction (this corresponds to treating E as a directed set of edges, so the development in the remainder of Section 3.2 is applicable to the more general setting in which G is a directed graph).

The state of processor p_i in a system state is drawn from Σ_i, the sequence of states p_i goes through as the distributed computation progresses, and is denoted by σ_i. Similarly, the state of a channel (p_i, p_j) is simply a set of messages, representing the messages that are *in transit* from p_i to p_j in that system state, i.e., messages that have been sent by p_i on channel (p_i, p_j) but not yet received by p_j. We denote this set by Φ_{ij}. The notion of a system state is very weak, in that it allows absurd global situations to be represented. For example, there is nothing in the definition of a system state that precludes the description of a situation in which a message φ has been sent by p_i on channel (p_i, p_j), but nevertheless neither has arrived at p_j nor is in transit on (p_i, p_j).

Our first definition of a global state is based on the partial order $\prec^+$ that exists on the set Ξ of events of the distributed computation, and requires the extension of $\prec^+$ to yield a *total order*, i.e., a partial order that includes exactly one of (ξ_1, ξ_2) or (ξ_2, ξ_1) for all $\xi_1, \xi_2 \in \Xi$. This total order does not contradict $\prec^+$, in the sense that it contains all pairs of events already in $\prec^+$. It is then obtained from $\prec^+$ by the inclusion of pairs of concurrent events, that is, events that do not relate to each other according to $\prec^+$, in such a way that the resulting relation is indeed a partial order. A total order thus obtained is said to be *consistent* with $\prec^+$ (Figure 3.3).

Given any total order $<$ on Ξ, exactly $|\Xi| - 1$ pairs $(\xi_1, \xi_2) \in <$ can be identified such that every event $\xi \neq \xi_1, \xi_2$ is either such that $\xi < \xi_1$ or such that $\xi_2 < \xi$.



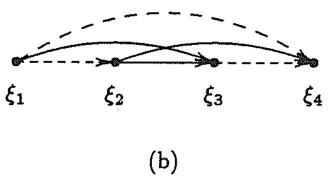


Figure 3.3. *Part (a) of this figure shows the precedence graph whose edge set, represented by full lines, is the partial order $\prec$. As $\prec$ is already transitive, we have $\prec^+=\prec$. Members of $\prec^+$ are then represented by full lines, while the dashed lines are used to represent the pairs of concurrent events, which, when added to $\prec^+$, yield a total order $\prec_t^+$ consistent with $\prec^+$. The same graph is redrawn in part (b) of the figure to emphasize the total order. In this case, $system\_state(\xi_2, \xi_3)$ is such that p_1 is in the state at which it was left by the occurrence of ξ_1, p_2 is in the state at which it was left by the occurrence of ξ_2, and a message sent in connection with ξ_2 is in transit on the channel from p_2 to p_1 to be received in connection with ξ_3. Because $\prec_t^+$ is consistent with $\prec^+$, $system\_state(\xi_2, \xi_3)$ is a global state, by our first definition of global states.*

Events ξ_1 and ξ_2 are in this case said to be *consecutive* in $<$. It is simple to see that, associated with every pair (ξ_1, ξ_2) of consecutive events in $<$, there is a system state, denoted by $system\_state(\xi_1, \xi_2)$, with the following characteristics.

(i) For each processor p_i, σ_i is the state resulting from the occurrence of the most recent event at p_i, say ξ, such that $\xi_1 \not< \xi$;

(ii) For each channel (p_i, p_j), Φ_{ij} is the set of the messages sent in connection with an event ξ such that $\xi_1 \not< \xi$ and received in connection with an event ξ' such that $\xi' \not< \xi_2$.

The first definition we consider for a global state is then the following. A

system state Ψ is a global state if and only if either in Ψ all processors are in their initial states (and then all channels are empty), or in Ψ all processors are in their final states (and then all channels are empty as well), or there exists a total order $\prec_t^+$, consistent with $\prec^+$, in which a pair (ξ_1, ξ_2) of consecutive events exists such that $\Psi = system\_state(\xi_1, \xi_2)$ (Figure 3.3).

Our second definition of a global state is somewhat simpler, and requires that we consider a partition of the set of events Ξ into two subsets Ξ_1 and Ξ_2. Associated with the pair (Ξ_1, Ξ_2) is the system state, denoted by $system\_state(\Xi_1, \Xi_2)$, in which σ_i is the state in which p_i was left by the most recent event of Ξ_1 occurring at p_i, and Φ_{ij} is the set of messages sent on (p_i, p_j) in connection with events in Ξ_1 and received in connection with events in Ξ_2.

The second definition is then the following. A system state Ψ is a global state if and only if $\Psi = system\_state(\Xi_1, \Xi_2)$ for some partition (Ξ_1, Ξ_2) of Ξ such that

$$Past(\xi) \subseteq \Xi_1$$

whenever $\xi \in \Xi_1$. (Equivalently, we might have required the existence of a partition (Ξ_1, Ξ_2) such that

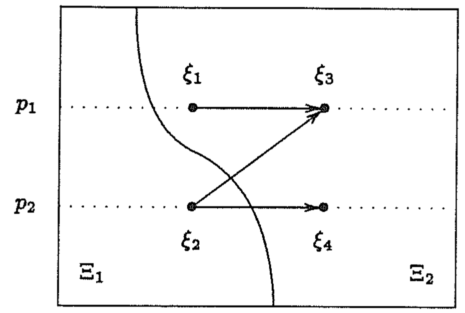
$$Future(\xi) \subseteq \Xi_2$$

whenever $\xi \in \Xi_2$.) Note that there is no need, in this definition, to mention explicitly the cases in which all processors are either in their initial or final states, as we did in the case of the first definition. These two cases correspond, respectively, to $\Xi_1 = \emptyset$ and $\Xi_2 = \emptyset$.

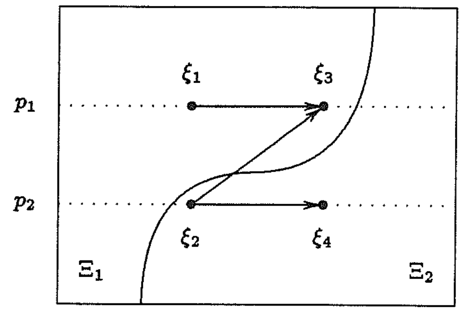
It is simple to understand that our two definitions of a global state are equivalent to each other. The first definition, however, is more suitable to the proof of Theorem 3.2 we present in Section 3.2.2, whereas the second definition provides us with a more intuitive understanding of what a global state is. Specifically, the partition (Ξ_1, Ξ_2) involved in this definition can be used in connection with the precedence graph H introduced in Section 3.1.3 to yield the following interpretation. The partition (Ξ_1, Ξ_2) induces in H a cut (a set of edges) comprising edges that lead from events in Ξ_1 to events in Ξ_2 and edges from events in Ξ_2 to events in Ξ_1. If $system\_state(\Xi_1, \Xi_2)$ is a global state, then this cut contains no edges from Ξ_2 to Ξ_1, and then comprises the edges that represent the local states of all processors (except those in their initial or final states) in that global state, and the edges that represent messages in transit in that global state (Figure 3.4).

It should be noted that the concept of a global state is equally applicable in both the synchronous and the asynchronous models, although it is in the latter case that its importance is more greatly felt. For synchronous systems, many global states can be characterized in association with the value of the global clock, as for example in "the global state at the beginning of pulse $c \geq 0$." However, there is nothing in the definition of a global state that precludes the existence in synchronous systems of global states in which processors' local states include values of the global clock that differ from processor to processor.

Having defined a global state, we may then extend the definitions of the past and the future of an event, given in Section 3.1.3, to encompass similar notions



(a)



(b)

Figure 3.4. *In this figure, the same precedence graph is shown in parts (a) and (b). Two different cuts are shown, one in each part, each establishing a different partition (Ξ_1, Ξ_2) of Ξ. The cut in part (a) has no edge leading from an event in Ξ_2 to an event in Ξ_1, and then, by our second definition of global states, system_state(Ξ_1, Ξ_2) is a global state. In this global state, p_1 is in its initial state, p_2 is in the state at which it was left by the occurrence of ξ_2, and a message is in transit on the channel from p_2 to p_1, sent in connection with ξ_2 and to be received in connection with ξ_3. The cut in part (b), on the other hand, has an edge leading from $\xi_2 \in \Xi_2$ to $\xi_3 \in \Xi_1$, so system_state(Ξ_1, Ξ_2) cannot be a global state.*

with respect to global states. If Ψ is a global state, we define its *past* and *future* respectively as

$$\begin{aligned} Past(\Psi) &= \bigcup_{\xi \in \Xi_1} [\{\xi\} \cup Past(\xi)] \\ &= \Xi_1 \end{aligned}$$

and

$$\begin{aligned} Future(\Psi) &= \bigcup_{\xi \in \Xi_2} [\{\xi\} \cup Future(\xi)] \\ &= \Xi_2, \end{aligned}$$

where $\Psi = system\_state(\Xi_1, \Xi_2)$ (this definition demonstrates another situation in which our second definition of a global state is more convenient). Similarly, we say that a global state Ψ_1 comes *earlier* in the computation Ξ than another global state Ψ_2 if and only if $Past(\Psi_1) \subset Past(\Psi_2)$ (or, alternatively, if $Future(\Psi_2) \subset Future(\Psi_1)$). The definition of an *earliest* global state with certain characteristics in a computation can be given likewise.

To finalize this section, let us now return to the question raised at its beginning, namely of how to express time-related assertions in an asynchronous model. Regarding Algorithm *A_Pif*, we can now state that "all processors have received *inf* at the earliest global state in which $received_i =$ **true** for all $1 \leq i \leq n$." In the same vein, it is now worthwhile to return to the statement of Theorem 3.1 and recognize that it would have been more precise, had we had the appropriate notions at the time.

3.2.2. An algorithm for global state recording

The idea of a global state is very attractive for allowing time-related issues to be discussed properly in asynchronous systems. This is going to be all, however, unless a mechanism is devised that allows the recording of global states for use in the subsequent analysis of global properties. As we shall discuss later in this section and in Section 3.3.1, it is this possibility of discussing global properties over a "frozen" picture of the system (recall that a global state is also called a snapshot) that accounts for most of the practical interest in the concept of a global state.

Before we introduce an algorithm for global state recording, it should be noted that we are now dealing with two distributed parallel computations. One of them, called the *substrate*, is the computation whose global properties one wishes to study. It is to the substrate that the set of events Ξ refers. The other distributed parallel computation is the algorithm for *Global state recording* — *Gsr*, Algorithm *Gsr*. Both computations run on the same system, so each processor is responsible for executing its share of the substrate and of Algorithm *Gsr*. They are, however, totally independent of each other as far as causality relationships are concerned. Our only assumption about their interaction is that Algorithm *Gsr* is capable of "peeking" at the substrate's variables and messages with the purpose of recording a global state. Note that a processor participates in both computations in such a way that, when the substrate is being executed by a processor, Algorithm *Gsr* is suspended, and conversely. This is immaterial from the standpoint of both computations, as they were designed to operate in an asynchronous model, and the suspending of one to execute the other only adds to the asynchronism already present.

The following is an outline of how Algorithm *Gsr* functions. Processor p_i is responsible for recording the substrate's local state and the states of all channels incoming to p_i, i.e., (p_j, p_i) such that $n_j \in \mathcal{N}(n_i)$. If all processors carry their recording tasks to their ends, then the resulting overall recording is a system state, as a state has been recorded for each processor and a set of messages for each channel. Algorithm *Gsr* can be initiated spontaneously by any number of processors concurrently, and progresses by the exchange between neighbor processors of a special message called *marker*. A processor p_i initiates Algorithm *Gsr* spontaneously by recording the local state of the substrate, σ_i, and then sending *marker* on all channels that outgo from it, without however allowing the substrate to send any messages in the meantime (i.e., after the recording of the state and before the sending of *marker*). In practice, this can be achieved by "disabling interrupts" so that the processor will not be switched to the other computation while this is undesired. All other processors behave likewise upon receiving *marker* for the first time. Every message of the substrate received at p_i from a neighbor p_j after p_i has received the first *marker* (and consequently recorded a local state) and before p_i receives *marker* from p_j is added to the set of messages Φ_{ji} that represents the state of channel (p_j, p_i). The state of the channel on which *marker* was first received is then recorded as the empty set, so the system state recorded by Algorithm *Gsr* can be regarded as containing a forest of empty channels rooted at the processors that initiated the recording process spontaneously. Algorithm *Gsr terminates* at a processor when it has received *marker* on all incoming channels.

Before we proceed to a more thorough description and analysis of Algorithm *Gsr*, let us briefly discuss some of its properties, regardless of whether the system state it records is indeed a global state. Notice, first of all, that *marker* messages do reach all processors, and therefore Algorithm *Gsr* is executed and terminates in all of them. This is so because in G all edges are undirected, and consequently all communication channels are bidirectional. As we remarked in Section 3.2.1, however, our treatment in Section 3.2 is applicable to directed graphs as well, in which case G is required to be strongly connected for the proper functioning of Algorithm *Gsr*. This established, it is simple to see that Algorithm *Gsr* has a time complexity of $O(n)$ and a communication complexity of $O(|E|)$. The argument here is entirely similar to the one given in Section 3.1.4 when we discussed the complexity of Algorithms *S_Pif* and *A_Pif*.

In the description of Algorithm *Gsr* we give next, we assume that all channels are bidirectional, so the distinction between "incoming" and "outgoing" channels does not have to be made — *marker* messages are sent to and received from all neighbors. Also, we let P_1 denote the set of processors that initiate Algorithm *Gsr* spontaneously, and denote by *comp_msg* the messages of the substrate.

In Algorithm *Gsr*, processor p_i maintains a variable σ_i to store the substrate's state at p_i and a variable Φ_{ji}, for all $n_j \in \mathcal{N}(n_i)$, to store the state of channel (p_j, p_i). In addition, a variable $recorded_i$ indicates whether the substrate's state has already

been recorded locally, and a variable $received_i(j)$ for each $n_j \in \mathcal{N}(n_i)$ indicates whether *marker* has been received from p_j. Clearly, for $p_i \notin P_1$, $recorded_i =$ **true** if and only if $received_i(j) =$ **true** for some $n_j \in \mathcal{N}(n_i)$, but we do keep the redundancy for clarity of exposition.

```
Algorithm Gsr:

  ▷ Variables:
        $\sigma_i$ = nil;
        $\Phi_{ji} = \emptyset$ for all $n_j \in \mathcal{N}(n_i)$;
        $recorded_i$ = false;
        $received_i(j)$ = false for all $n_j \in \mathcal{N}(n_i)$.

  ▷ Input:
        $msg$ = nil.
    Action at $p_i \in P_1$:
        Set $\sigma_i$ to the current state of the substrate;
        $recorded_i$ := true;
        Send marker to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.

  ▷ Input:
        $msg$ = marker.
    Action:
        $received_i(j)$ := true for $p_j = origin_i(msg)$;
        if not $recorded_i$ then
            begin
                Set $\sigma_i$ to the current state of the substrate;
                $recorded_i$ := true;
                Send marker to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
            end.

  ▷ Input when $recorded_i$ and not $received_i(j)$ for some $n_j \in \mathcal{N}(n_i)$:
        $msg$ = comp_msg.
    Action:
        if not $received_i(j)$ for $p_j = origin_i(msg)$ then
            $\Phi_{ji} := \Phi_{ji} \cup \{msg\}$ for $p_j = origin_i(msg)$.
```

There are two important notational observations to be made concerning Algorithm *Gsr*. The first observation is that the assumed atomicity of actions in our distributed algorithm notation (see Section 3.1.5) suffices to prevent the processor from executing the substrate computation, possibly with the sending of messages, between the recording of the local state and the sending of *marker*'s. Secondly, the last input/action pair is based on a conditional input, i.e., the input is only to be considered when at least one *marker* has already been received (or, if in P_1, the processor has already "waken up") and a *marker* is expected from at least one neighbor. While the inclusion of such conditions fits well in the usual practice of concurrent programming with the use of *guarded commands* (see Appendix A), it is not necessary in the description of most algorithms in this book. In the case of Algorithm *Gsr*, however, it has been instrumental in indicating that the last input/action pair is only to be considered while the recording process has not terminated locally. As *msg* is in this case a message of the substrate, the conditional input has been used to indicate that such messages are only to be looked at by Algorithm *Gsr* until the recording is over.

The following theorem establishes the correctness of Algorithm *Gsr* when every communication channel is a FIFO (First In, First Out) channel, i.e., it delivers the messages that go through it in the order they are sent.

Theorem 3.2. *If all communication channels are FIFO, then the system state recorded by Algorithm Gsr is a global state.*

Proof: Let (Ξ_1, Ξ_2) be a partition of Ξ such that $\xi_1 \in \Xi_1$ if and only if ξ_1 occurred before the state of the processor at which it occurred was recorded. In addition, let $\prec_t^+$ be any total order consistent with $\prec^+$, and consider two consecutive events $\xi_2 \in \Xi_2$ and $\xi_1 \in \Xi_1$ in $\prec_t^+$.

By the definition of Ξ_1 and of Ξ_2, it is clear that ξ_2 did not happen at the same processor as ξ_1 and before the occurrence of ξ_1. Now consider a scenario in which a sequence of events follow ξ_2 at the processor at which it happened, and then a message is sent in connection with the last event in this sequence, which in turn eventually causes the sending of another message by its destination processor, and then the eventual sending of another message by the destination processor of this second message, and so on, and then the arrival of the last message causes a sequence of events to happen at its destination processor culminating with the occurrence of ξ_1. By Algorithm *Gsr*, and by the definition of Ξ_1 and Ξ_2, the processor at which ξ_2 happened must have sent *marker*'s before ξ_2 happened. Likewise, the processor at which ξ_1 happened must not have received any *marker* before ξ_1 happened. Clearly, these two requirements are inconsistent with the scenario we just described, as the communication channels are all FIFO, and the sequence of messages alluded to in the description of the scenario would then have to have been overrun by a *marker* (Figure 3.5). In summary, $(\xi_2, \xi_1) \notin \prec^+$, so the total order $\prec_t^+$ can be altered by substituting (ξ_1, ξ_2) for (ξ_2, ξ_1) in it, and yet remain consistent with $\prec^+$.

Clearly, it takes no more than $|\Xi_1||\Xi_2|$ such substitutions to obtain a total order in which at most one pair (ξ_1, ξ_2) of consecutive events exists such that $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$. The events in all other pairs of consecutive events are in this total order both in Ξ_1 or in Ξ_2. By the definition of Ξ_1 and Ξ_2, this distinguished pair of consecutive events is such that *system_state*(ξ_1, ξ_2) is precisely the system state recorded by Algorithm *Gsr*, which is then a global state, by our first definition of global states. ■

In Section 3.3.1, we discuss the use of Algorithm *Gsr* in the analysis of important system-wide issues, particularly the one of termination.

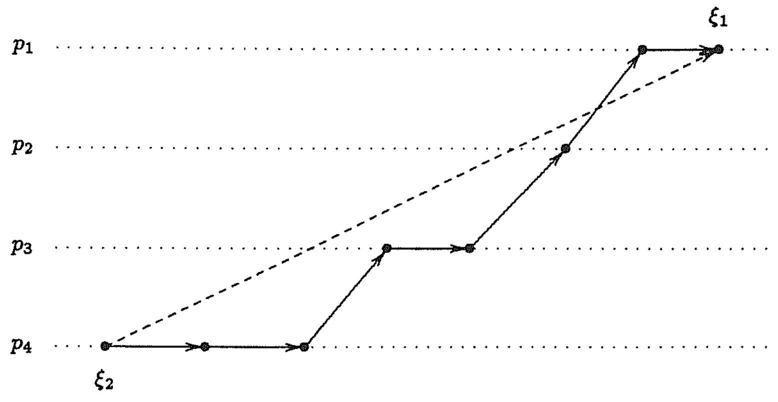


Figure 3.5. *In this fragment of a precedence graph, edges in $\prec$ are shown in full lines. If ξ_2 happened at p_4 after it recorded its state and ξ_1 happened at p_1 before it recorded its state, then this precedence graph fragment must not have been generated, as the three messages involved in it would have to have been overrun on the FIFO channels by a sequence of marker's, each sent as a consequence of the reception of another, starting with the marker sent by p_4 before the occurrence of ξ_2, and ending with the marker received by p_1 after the occurrence of ξ_1. The pair of events (ξ_2, ξ_1), shown in a dashed line, cannot then be in $\prec^+$.*

3.3. TERMINATION DETECTION

3.3.1. The general case

The issue of distributed algorithm termination is not new to us, as we have already discussed the question of local termination in the context of Algorithms *A_Pif* and *Gsr* in Sections 3.1.4 and 3.2.2, respectively. Our interest now is to discuss the termination of distributed algorithms in a global perspective, and this, as one may suspect, will require additional algorithmic techniques.

A distributed algorithm is said to have *terminated globally*, or simply *terminated*, when all processors are in their final states in that computation (and then no messages are in transit in any of the communication channels). Clearly, at such a stage the computation progresses no further, as no more messages can be delivered to trigger the occurrence of events.

The issue of global termination is of fundamental importance in both synchronous and asynchronous systems. In the case of synchronous systems, however, it is often the case that the model's peculiarities can be used to aid in the detection of a termination condition. Recall, for example, Algorithm *S_Pif*, given in Section 3.1.5 for the synchronous propagation of information with feedback. In this algorithm, the processor originating the propagation can be sure of a global termination condition by simply waiting for the appropriate amount of clock pulses to elapse. As we already know, no such simple mechanism can be expected to work in an asynchronous environment. In the remainder of Section 3.3, we shall then concentrate on the discussion of techniques for the detection of termination in asynchronous systems. The results are, of course, applicable to synchronous systems as well.

The definition of global termination we have just given is of course intimately related to the notion of global states we discussed in Section 3.2, as it can be rephrased to emphasize that the computation has terminated globally at a certain global state if and only if in that global state all processors are in their final states, and, consequently, all channels are empty. In fact, global termination is one of the so-called *stable properties* of a distributed computation. Another stable property of great interest is *deadlock*, to be discussed in the context of Section 4.2 and later in Appendix A. Informally, a group of processors is said to be in a condition of deadlock when every processor in the group is waiting for a condition that can only be caused by processors in the same group.

A property of a distributed computation is said to be stable if and only if, when it holds for a global state Ψ, then it holds for all global states that come later than Ψ. This is clearly the case of global termination and of deadlock.

We then have a general framework for detecting the occurrence of stable properties of distributed computations in general, and of global termination in particular. A processor wishing to check whether the property holds initiates a global state recording, by means of Algorithm *Gsr* of Section 3.2.2 or any other equivalent technique. After the global state has been recorded, every processor in the system sends its share of the recorded global state to the processor that initiated the recording (this can be done immediately upon local termination of the recording at each of the processors). The initiating processor, upon receiving such reports from all other processors, assembles the pieces to yield a meaningful global state and checks whether the desired property holds for that global state. If it does, then the processor knows that the property also holds for all global states coming later than the one recorded. If it does not, then no definite statement can be made regarding the property, as it may or may not hold currently in the system. Additional global states will then have to be recorded for further investigation. (In fact, when the property does not hold for the recorded global state, all that can be said is that it does not hold for all global states coming earlier than that one either, by the definition of a stable property.)

This process of transferring pieces of information from all processors with a common destination (in this case the processor that initiated the stable property detection) relies on mechanisms for message routing we have not so far encountered in this book, as we have only been dealing with messages between neighbors. We shall in Appendix A provide a more thorough discussion of this topic.

The generic method we have discussed in this section for the detection of stable properties in general (global termination in particular) may seem unpredictably

wasteful, inasmuch as it requires a number of attempts (each involving a global state recording) that is hard to anticipate. Nevertheless, for many applications this general technique can be adapted to yield satisfactory results, often utilizing an additional processor, say p_0, whose sole function is the treatment of the assembled global states for the detection of termination. More will be said about this scheme in Section 4.3, when we discuss the generic algorithms for simulating the automaton networks that we treat in this book.

3.3.2. The case of diffusing computations

While the scheme we discussed in Section 3.3.1 is applicable to the detection of global termination in general, some classes of distributed computations lend themselves quite well to more specialized techniques that may, depending on the case, be more efficient. One such class of computations is that of the so-called *diffusing computations*.

A distributed computation is said to be a diffusing computation if and only if it is initiated by a single processor. This is, for example, the case of the algorithm we saw in Section 3.1.5 for the asynchronous propagation of information with feedback, namely Algorithm *A_Pif*. In the case of this algorithm, however, detecting global termination was a straightforward matter, mainly because of the computation's great simplicity. Other diffusing computations exist, however, that do not lend themselves so well to a simple method of global termination detection.

The discussion we present in this section is tailored to the detection of termination of diffusing computations, but can nevertheless be applied to the detection of termination of other types of computation as well. This is achieved by considering an additional processor, say p_0, whose function is to serve as the sole initiator of the computation, and to detect its termination. This processor is connected to the processors that originally functioned as initiators, and starts its participation in the computation by sending messages to those processors so they can initiate the computation themselves. From this point onward, the participation of p_0 in the computation is restricted to the aspects related to the detection of global termination. Notice that this artifice imposes no limitations on the asynchronous computations under consideration, as we have simply replaced the spontaneous initiation by some processors with the sending to those processors by p_0 of initial start-up messages. We shall in the remainder of this section assume the presence of such a processor p_0.

One may suspect at this point that the detection of termination for diffusing computations will operate on a substrate computation, as we did with the recording of global states in Section 3.2.2, especially in view of the generic techniques we discussed in Section 3.3.1 for termination detection. The algorithm for global state recording, Algorithm *Gsr*, operates in such a way that the substrate is not causally affected by its presence, nor is it affected by the substrate's — Algorithm *Gsr* operates as a "probe," so its own functioning and termination do not depend on the substrate. In the case of detecting the termination of diffusing computations, nevertheless, there are peculiarities that suggest that this view of the two computations is not the most adequate, as the detecting algorithm functions more as a "modifier" than as a "probe" (this is also the case with other similarly important techniques, as for example the synchronizers we discuss in Section 4.1).

We assume that in the original diffusing computation (the one to be modified with the purpose of termination detection) processor p_i responds to the reception of a message *comp_msg* by calling a procedure COMPUTE$_i$(*comp_msg*), where $1 \leq i \leq n$. This assumption includes the cases in which *comp_msg* is one of the special messages sent by p_0 to initiate the computation. The modified algorithm then has the following general appearance. Every *comp_msg* is acknowledged with an *ack* message. For $0 \leq i \leq n$, processor p_i maintains a counter $expected_i$ to indicate the number of *ack* messages it expects from its neighbors (we assume for $1 \leq i \leq n$ that COMPUTE$_i$ automatically increases $expected_i$ whenever it sends a *comp_msg*). For $1 \leq i \leq n$, p_i also maintains, as in the case of Algorithm *A_Pif*, a variable $parent_i$ to indicate the origin of a *comp_msg* received in a special situation to be described shortly. The behavior of processors $p_1, \ldots, p_n$ is as follows. Whenever p_i receives a *comp_msg* and $expected_i > 0$, an *ack* is immediately sent in response. If, on the other hand, a *comp_msg* is received and $expected_i = 0$, then the *ack* is withheld and sent only when $expected_i$ becomes equal to zero again (if it at all changes with the execution of COMPUTE$_i$(*comp_msg*); otherwise the *ack* is sent immediately after the execution of COMPUTE$_i$). The variable $parent_i$ is in this case set to contain the identity of the neighbor that sent *comp_msg* until the *ack* can be sent. The algorithm *terminates* locally at p_i when $expected_i$ becomes zero and the pending *ack*, if any, is sent to $parent_i$. This condition may, however, change many times during the computation, for $expected_i$ may again acquire a positive value as a consequence of the reception of a *comp_msg*. The behavior of p_0 is significantly simpler, as it never receives a *comp_msg*. When $expected_0$ reaches the value zero, the computation has terminated globally.

The combined algorithm is presented next under the name of Algorithm *Diff*. It is given in two versions, one for p_0 and one for $p_1, \ldots, p_n$. As in Section 3.2.2, we let P_1 denote the set of processors that in the original computation were the initiators. They will in the modified algorithm be the ones to receive special initiating messages from p_0.

```
Algorithm Diff for $p_0$:
 ▷ Variables:
       $expected_0 = 0$.
 ▷ Input:
       $msg = \mathbf{nil}$.
   Action:
       Send comp_msg to every $p_j \in P_1$;
       $expected_0 := expected_0 + |P_1|$.
 ▷ Input:
       $msg = ack$.
   Action:
       $expected_0 := expected_0 - 1$.

Algorithm Diff for $p_i \in \{p_1, \ldots, p_n\}$:
 ▷ Variables:
       $expected_i = 0$;
       $parent_i = \mathbf{nil}$.
 ▷ Input:
       $msg = comp\_msg$.
   Action:
       if $expected_i > 0$ then
           begin
               Send ack to $origin_i(msg)$;
               COMPUTE$_i(msg)$
           end
       else
           begin
               COMPUTE$_i(msg)$;
               if $expected_i > 0$ then
                   $parent_i := origin_i(msg)$
               else
                   Send ack to $origin_i(msg)$
           end.
 ▷ Input:
       $msg = ack$.
   Action:
       $expected_i := expected_i - 1$.
       if $expected_i = 0$ then
           Send ack to $parent_i$.
```

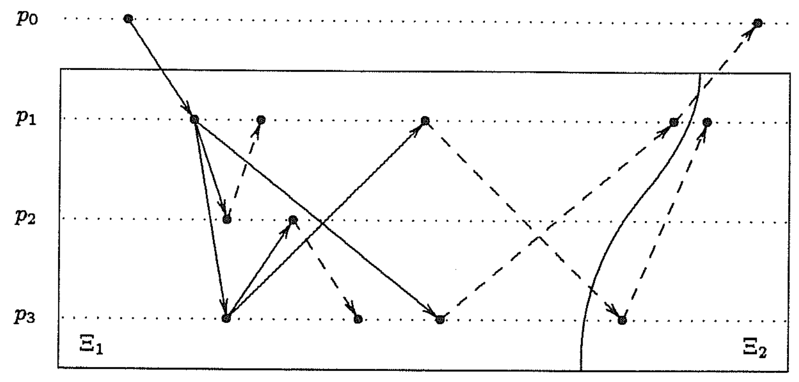
The following theorem establishes the correctness of Algorithm *Diff*.

Theorem 3.3. *If the diffusing computation terminates, then Algorithm Diff terminates at all processors, and upon termination at p_0 it has also terminated globally.*

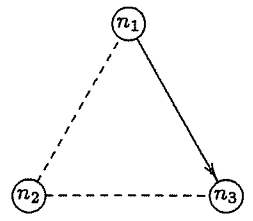
Proof: If the diffusing computation terminates, then every processor sends a finite number of *comp_msg*'s. As these *comp_msg*'s and the corresponding *ack*'s are received, the value of $expected_i$ for processor p_i, initially equal to zero, becomes positive and zero again, possibly several times. When a transition occurs in the value of $expected_i$ from zero to a positive value, $parent_i$ is set to the identity of the processor that sent the corresponding *comp_msg*. Consider the system states in which every processor is either in a state of positive $expected_i$ following the last transition from zero of its value, if it ever sent a *comp_msg* during the diffusing computation, or in any state, otherwise. Clearly, at least one of these system states is a global state (cf. our definitions in Section 3.2.1), as for example the one in which every processor that ever sent *comp_msg*'s is in its state that immediately precedes the reception of the last *ack* (Figure 3.6). In this global state, only *ack*'s flow on the channels, none of which sent as a consequence of the reception of a last *ack*. Let us consider one of these global states, and let $P_1' \subseteq P_1$ contain the processors in P_1 that ever sent a *comp_msg*.

In this global state, the variables $parent_i$ for $p_i \notin P_1'$ induce a forest that spans all nodes in G corresponding to processors that sent at least one *comp_msg* during the diffusing computation. (This forest is in fact dynamically changing with the progress of Algorithm *Diff*, as $parent_i$ may point to several of p_i's neighbors along the way; it is always a forest, nevertheless.) The trees in this forest can be thought of as being rooted at the nodes corresponding to the processors in P_1', and then their leaves correspond to those processors from which no other processor p_i received the *comp_msg* that triggered the last transition from zero to a positive value of $expected_i$ (Figure 3.6). As in the proof of Theorem 3.1, we proceed by induction on the subtrees of the trees in the forest. Along the induction, the assertion to be shown is that Algorithm *Diff* terminates at all processors corresponding to nodes in the subtree, and that upon termination at the processor corresponding to the subtree's root Algorithm *Diff* has also terminated at every processor corresponding to nodes in the subtree.

For each tree, the basis of the induction is given by the subtrees rooted at the leaves, and then the assertion clearly holds, as no leaf n_i is such that $p_i = parent_j$ for some processor p_j. As the induction hypothesis, assume the assertion for all the subtrees rooted at nodes corresponding to processors p_j such that $parent_j$ is $p_k \in P_1'$. Then p_k receives $expected_k$ *ack*'s, at which time Algorithm *Diff* has terminated at all processors corresponding to nodes in the subtree rooted at n_k. The theorem then follows by the observation that this holds for all trees, so every processor $p_k \in P_1'$ sends an *ack* to p_0. Upon the reception of these *ack*'s, Algorithm *Diff* terminates at p_0, and at this time it has also terminated globally. ■



(a)



(b)

Figure 3.6. *Edges in the precedence graph fragment shown in part (a) are drawn as either full lines or dashed lines. Full lines represent comp_msg's, dashed lines represent ack's, and the remaining edges of the precedence graph (those connecting events within the same processor) are omitted. In this case, system_state(Ξ_1, Ξ_2) is clearly a global state, and is such that every processor that ever sent a comp_msg during the diffusing computation (i.e., p_1 and p_3) is in its state that immediately precedes the reception of the last ack. In part (b), the spanning forest formed by the variables $parent_i$ for each processor p_i in this global state is shown with directed edges that point from n_i to n_j to indicate that $parent_i = p_j$. In this case, $P_1' = P_1 = \{p_1\}$, and then the forest has one single tree whose root is n_1 and whose single leaf is n_3.*

3.4. BIBLIOGRAPHIC NOTES

Various books on parallel computing exist that should be useful in the broad context discussed in the beginning of Section 3.1, as for example Hillis (1985), Fox, Johnson, Lyzenga, Otto, Salmon, and Walker (1988), and Almasi and Gottlieb (1989). In addition, there are also the articles by Seitz (1985) and by Bell (1992).

A wealth of models exist for distributed parallel computations. Good references on these models and developments thereof are Pnueli (1981), Lynch and Tuttle (1987), Chandy and Misra (1988), Chou and Gafni (1988), Welch, Lamport, and Lynch (1988), Lynch and Goldman (1989), and Manna and Pnueli (1992). The synchronous and asynchronous models adopted in this book can be looked up in Lamport and Lynch (1990), and are compared in an interesting article by Arjomandi, Fischer, and Lynch (1983). Other models of interest, including the PRAM, are discussed, with algorithms, by Gibbons and Rytter (1988), Akl (1989), Karp and Ramachandran (1990), JáJá (1992), and Leighton (1992). A chapter on the subject is provided by Cormen, Leiserson, and Rivest (1990). An innovative and more recent proposal can be found in Feldman and Shapiro (1992).

The event-based formalism of Section 3.1 is based on Lamport (1978), and has benefited from the clearer view given by Bracha and Toueg (1984). The complexity measures we have adopted are of common use in many places, being also discussed by Lynch and Goldman (1989) and Lamport and Lynch (1990). The Pif problem is discussed by Segall (1983). Our distributed algorithm notation, based on the use of input/action pairs, follows the general guidelines of guarded commands (Dijkstra, 1975), and is therefore aimed at its further use in the book.

For a discussion on the limitations arising from the assumption of an unknown value of n, the reader should refer to Angluin (1980). The effects of the assumption of an anonymous system are discussed by Attiya, Snir, and Warmuth (1988) and by Attiya and Snir (1991).

The material presented in Section 3.2 on distributed snapshots, their properties, and a recording algorithm, is based on Chandy and Lamport (1985).

The discussion in Section 3.3 on termination detection is partly based on Bracha and Toueg (1984) and Chandy and Lamport (1985), where distributed snapshots are employed for the detection of stable properties. The detection of termination for diffusing computations is based on Dijkstra and Scholten (1980), and can also be found in Bertsekas and Tsitsiklis (1989).

4 Timing and synchronization

This chapter continues the treatment, at an abstract level, of the material presented in Chapter 3 on the design and analysis of distributed algorithms. The emphasis is now, however, on the design of distributed parallel algorithms for the simulation of FC and PC automaton networks. The basic techniques for this simulation are discussed in Sections 4.1 (synchronizers, used for FC automaton network simulation) and 4.2 (scheduling by edge reversal, used for PC automaton network simulation). Template algorithms for automaton network simulation based on these techniques are given in Section 4.3. Bibliographic notes follow in Section 4.4.

4.1. SYNCHRONIZERS

4.1.1. Synchronizing algorithms

We have seen in most of Section 3.1 that an interesting tradeoff exists between using a synchronous or an asynchronous model of distributed parallel computation in the design of algorithms. If, on the one hand, the synchronous model offers the simplicity of global synchronization and bounded communication delays, on the other hand the asynchronous model, with its absence of a global time basis and unpredictable message delays, is far more realistic.

In all of Section 4.1, we discuss a generic algorithmic tool, called a *synchronizer*, that allows a distributed algorithm designed for a synchronous model to be "translated" into another distributed algorithm that executes correctly in an asynchronous model. Synchronizers are of great conceptual and practical interest in general, but are particularly relevant within the context of this book, because they provide the fundamental underpinnings of the algorithms we shall utilize in the distributed parallel simulation of the FC automaton networks we introduced in Chapter 1.

Just as in the case of the algorithm we described in Section 3.3.2 for the detection of termination of diffusing computations, a synchronizer acts as a "modifier" of the original synchronous algorithm. It is therefore more appropriate to refer to the resulting asynchronous algorithm as a single entity than to regard the synchronous algorithm as a substrate upon which the synchronizer acts (as we did in the case of Algorithm *Gsr* for global state recording in Section 3.2.2).

Throughout Section 4.1, we utilize the following notation. The original synchronous algorithm is called Algorithm *S_Alg*, and the asynchronous algorithm resulting from the application of the synchronizer Algorithm *A_Alg*. Algorithm *S_Alg*, as a synchronous algorithm, proceeds in pulses, and at the beginning of each pulse processor p_i executes a procedure $\textsc{Compute}_i(c, MSG)$. Following the notation introduced in Section 3.1.5, $c \geq 0$ identifies the pulse and for $c > 0$ MSG is the set of messages received by p_i during pulse $c - 1$. MSG is empty when $c = 0$.

The essential property that we seek to preserve in translating Algorithm *S_Alg* into Algorithm *A_Alg* is that no processor will proceed to pulse $c + 1$ before all messages sent to it during pulse c have been delivered and incorporated into MSG. In order to ensure that this property holds for all processors and at all clock pulses, we begin by requiring that all messages of Algorithm *S_Alg* be acknowledged. These messages are, as in Sections 3.2.2 and 3.3.2, denoted by *comp_msg*, and the acknowledgements by *ack*. A processor is said to be *safe* with respect to pulse $c \geq 0$ if and only if it has received an *ack* for every *comp_msg* it sent during pulse c. In order to guarantee that our essential property holds for p_i at pulse c, it then suffices that p_i receive information stating that every one of its neighbors is safe with respect to pulse c. The task of a synchronizer is then to convey this information to all processors concerning all pulses of the synchronous computation.

Let *Sync* denote a generic synchronizer, to be understood as a distributed algorithm that is repeated at every pulse of Algorithm *S_Alg* in order to convey to all processors the safety information we have identified as fundamental. Then, employing the notation introduced in Section 3.1.5 to describe the communication and time complexities of distributed algorithms, we have that $Comm(Sync)$ and $Time(Sync)$ stand for the communication and time (in the asynchronous sense) overheads, respectively, introduced by *Sync* per pulse of Algorithm *S_Alg* to yield Algorithm *A_Alg*.

Regardless of how *Sync* operates, we can already draw some conclusions regarding the final complexity of Algorithm *A_Alg*. Let us, first of all, recognize that the use of the *ack* messages does not add to the communication complexity of Algorithm *S_Alg*, as exactly one *ack* is sent per *comp_msg*. Considering in addition that $Comm(Sync)$ is the communication overhead introduced by *Sync* per pulse of the execution of Algorithm *S_Alg*, and that there are $Time(S\_Alg)$ such pulses, we then have

$$Comm(A\_Alg) = Comm(S\_Alg) + Time(S\_Alg)Comm(Sync) + Comm_0(Sync),$$

where $Comm_0(Sync)$ is the communication cost, if any, that *Sync* incurs with initialization procedures.

Similarly, as $Time(Sync)$ is the time overhead (in the asynchronous sense) introduced by *Sync* per each of the $Time(S\_Alg)$ pulses of Algorithm *S_Alg*, we

have

$$Time(A\_Alg) = Time(S\_Alg)\,Time(Sync) + Time_0(Sync),$$

where $Time_0(Sync)$ refers to the time, if any, needed by *Sync* to be initialized.

Depending on how *Sync* is designed, the resulting $Comm(A\_Alg)$ and $Time(A\_Alg)$ can vary considerably. In Section 4.1.2, we discuss three types of synchronizers, with particular emphasis on the one that will be used in the simulations discussed in this book.

4.1.2. Some synchronizers

We saw in Section 4.1.1 that the essential task of a synchronizer is to convey to every processor at every pulse the information that all of the processor's neighbors are safe with respect to that pulse. This safety information indicates that the processor's neighbors have received an *ack* for every *comp_msg* they sent in the current pulse, and therefore the processor may proceed to the next pulse.

The first synchronizer we present is known as the synchronizer *Alpha*. It is the most important synchronizer within the scope of this book, inasmuch as one of its variants contains the mechanisms needed to simulate FC automaton networks. In *Alpha*, the information that all of a processor's neighbors are safe with respect to a pulse $c \geq 0$ is conveyed directly by each of those neighbors by means of a $safe(c)$ message. A processor may then proceed to pulse $c+1$ when it has received a $safe(c)$ from each of its neighbors. Clearly, we have

$$Comm(Alpha) = O(|E|)$$

and

$$Time(Alpha) = O(1),$$

as a *safe* message is sent between each pair of neighbors in each direction, and causes no effect that propagates farther than one hop away for the duration of one pulse. We also have $Comm_0(Alpha) = Time_0(Alpha) = 0$.

We now give a detailed description of Algorithm *A_Alg* when obtained from Algorithm *S_Alg* by means of *Alpha*. The algorithm we describe is called Algorithm *A_Alg(Alpha)*. Our description does not assume at first that communication channels are FIFO, and for this reason *comp_msg*'s and *ack*'s sent in connection with pulse $c \geq 0$ are sent as $comp\_msg(c)$ and $ack(c)$. In this algorithm, processor p_i maintains a variable $cp_i \geq -1$ to indicate, if nonnegative, the pulse of Algorithm *S_Alg* it is currently executing. Notice that it is now fundamental that p_i keeps track of the value of this variable, as the global clock needed by Algorithm *S_Alg* no longer exists. A variable $MSG_i(c)$, in addition, is used by p_i for each $c \geq 0$ to store the set of messages to be consumed by $\textsc{Compute}_i$ at pulse c. Another variable used by p_i in connection with pulse $c \geq 0$ is $expected_i(c)$, which records the number of $ack(c)$'s expected. As in our description of Algorithm *Diff* in Section 3.3.2, here too we assume that $\textsc{Compute}_i$ automatically increases the value of $expected_i(c)$ whenever it sends a $comp\_msg(c)$. Processor p_i also maintains a variable $safe_i(j, c)$

for each neighbor p_j and all $c \geq 0$, used to indicate whether a *safe*(c) has been received from p_j.

As in previous occasions, we let P_1 denote the set of processors that initiate the computation spontaneously. Upon doing so, the processors in P_1 will send a special message, *startup*, to all of its neighbors. This message, when received for the first time by any other processor, is forwarded to all of its neighbors. Loosely, this *startup* message can be though of as a "*safe*(-1)" message. All processors, including those in P_1, only proceed to executing pulse $c = 0$ of the synchronous computation upon receiving a *startup* from every neighbor. This is controlled by a variable $go_i(j)$, maintained by p_i for every neighbor p_j to indicate whether a *startup* has been received from p_j. An additional variable, up_i, indicates whether p_i has already begun the asynchronous computation, either spontaneously or after receiving the first *startup*.

```
Algorithm A_Alg(Alpha):

  ▷ Variables:
        $cp_i = -1$;
        $MSG_i(c) = \emptyset$ for all $c \geq 0$;
        $up_i$ = false;
        $go_i(j)$ = false for all $n_j \in \mathcal{N}(n_i)$;
        $expected_i(c) = 0$ for all $c \geq 0$;
        $safe_i(j, c)$ = false for all $n_j \in \mathcal{N}(n_i)$ and all $c \geq 0$.

  ▷ Input:
        $msg$ = nil.
    Action at $p_i \in P_1$:
        $up_i$ := true;
        Send startup to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.

  ▷ Input:
        $msg$ = startup.
    Action:
        $go_i(j)$ := true for $p_j = origin_i(msg)$;
        if not $up_i$ then
            begin
                $up_i$ := true;
                Send startup to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
            end;
        if $go_i(j)$ for all $n_j \in \mathcal{N}(n_i)$ then
            begin
                $cp_i := cp_i + 1$;
                COMPUTE$_i(cp_i, MSG_i(cp_i))$;
                if $expected_i(cp_i) = 0$ then
                    Send safe$(cp_i)$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
            end.
```

```
  ▷ Input:
        $msg = comp\_msg(c)$.
    Action:
        $MSG_i(c+1) := MSG_i(c+1) \cup \{msg\}$;
        Send ack$(c)$ to $origin_i(msg)$.

  ▷ Input:
        $msg = ack(c)$.
    Action:
        $expected_i(c) := expected_i(c) - 1$;
        if $expected_i(c) = 0$ then
            Send safe$(c)$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.

  ▷ Input:
        $msg = safe(c)$.
    Action:
        $safe_i(j, c)$ := true for $p_j = origin_i(msg)$;
        if $safe_i(j, cp_i)$ for all $n_j \in \mathcal{N}(n_i)$ then
            begin
                $cp_i := cp_i + 1$;
                COMPUTE$_i(cp_i, MSG_i(cp_i))$;
                if $expected_i(cp_i) = 0$ then
                    Send safe$(cp_i)$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
            end.
```

Algorithm *A_Alg*(*Alpha*), as presented, has a serious drawback, especially when it comes to a practical implementation. The problem is that many of the messages and variables it employs depend on the pulse number c, which in principle is an unbounded nonnegative integer. Clearly, these are troublesome circumstances for the usage of both communication channels and processors' memories. Although the presentation of the algorithm has benefited in clarity from the use of such messages and variables, they cannot be used in practice. A similar problem exists with maintaining the variable cp_i, as it too is in principle unbounded. We shall, however, keep it even under the simplifying assumptions we make in the sequel, because it may in a generic computation be needed by COMPUTE$_i$. For our main application in this book, nevertheless, this variable will be unnecessary, as we discuss in Section 4.3.2.

Before we describe our simplifications and the resulting algorithm, however, let us consider the following. Suppose, for the sake of example, that processor p_i has two neighbors, p_j and p_k. Suppose further that p_j has only one neighbor (p_i) and that p_k has many neighbors. Consider the situation in which p_i has just become safe with respect to pulse c and then sends *safe*(c) to p_j and p_k. Processor p_i can only proceed to pulse $c + 1$ after receiving similar messages from its two neighbors. Suppose that such a message has been received from p_j but not from p_k (which depends on many more neighbors other than p_i to become safe at pulse c). It is possible at this moment that a *safe*$(c + 1)$ too is received from p_j before the *safe*(c) from p_k arrives, at which time p_i will have received two *safe* messages from p_j without the respective counterparts from p_k, and will therefore be unable to proceed to pulse $c+1$. It is simple to see, nevertheless, that p_j will at this time be unable to proceed to pulse $c+2$, as it now depends on a *safe*$(c+1)$ from p_i. If we compute the number of *safe* messages p_i has received since the beginning of the computation from its two neighbors, we will see that the numbers corresponding to p_j and p_k differ by no more than two. The particular topological situation we described was meant to help the understanding of this issue, but the maximum difference we just stated is true in general.

This relative "boundedness," together with the assumption that all communication channels are FIFO, allows various simplifications to be carried out on Algorithm *A_Alg*(*Alpha*). For one thing, no message or variable needs to depend explicitly on c any longer. In addition, under the FIFO channel assumption *ack*'s are no longer needed, and p_i may send *safe* messages to all of its neighbors immediately upon completion of COMPUTE$_i$. Such *safe* messages will certainly be delivered after the *comp_msg*'s sent by COMPUTE$_i$, indicating that every such message sent by p_i during the current pulse has already arrived.

We now present a version of Algorithm *A_Alg*(*Alpha*) in which channels are assumed to be FIFO, and, in addition, the following important assumption is made. At each clock pulse $c \geq 0$, COMPUTE$_i$ sends exactly one *comp_msg* to each of p_i's neighbors. These two assumptions allow *startup*, *ack* and *safe* messages to be done away with, and so render the variables $go_i(j)$, $expected_i$ and $safe_i(j)$ useless. The behavior of p_i is now considerably simpler: it starts upon receiving the first *comp_msg* (unless it belongs to P_1), and proceeds to the next pulse upon receiving exactly one *comp_msg* from each of its neighbors. However, it is still possible to receive two consecutive *comp_msg*'s from one neighbor without having received any *comp_msg* from another neighbor. This issue is essentially the same we discussed above concerning the reception of multiple *safe* messages from a same neighbor, and some control mechanism has to be adopted. What we need is, for each neighbor, a queue with one single position in which *comp_msg*'s received from that neighbor are kept until they can be incorporated into MSG_i. (From our previous discussion, it would seem that two-position queues are needed. However, we can think of MSG_i as containing the queue heads for all of p_i's queues.) We then let $queue_i(j)$ denote this queue at p_i for neighbor p_j. The new version we present is named after *Scheduling by alpha synchronization*, Algorithm *Sas*, for its relation with the applications to be discussed in Section 4.3.

```
Algorithm Sas:
  ▷ Variables:
        $cp_i = -1$;
        $MSG_i = \emptyset$;
        $up_i =$ false;
        $queue_i(j) =$ nil for all $n_j \in \mathcal{N}(n_i)$.
  ▷ Input:
        $msg =$ nil.
    Action at $p_i \in P_1$:
        $up_i :=$ true;
        $cp_i := cp_i + 1$;
        COMPUTE$_i(cp_i, MSG_i)$.
  ▷ Input:
        $msg = comp\_msg$.
    Action:
        if not $up_i$ then
            begin
                $up_i :=$ true;
                $cp_i := cp_i + 1$;
                COMPUTE$_i(cp_i, MSG_i)$
            end;
        if there exists $msg' \in MSG_i$ such that $origin_i(msg') = origin_i(msg)$
        then
            $queue_i(j) := msg$ for $p_j = origin_i(msg)$
        else
            $MSG_i := MSG_i \cup \{msg\}$;
        if there exists $msg' \in MSG_i$ for all $n_j \in \mathcal{N}(n_i)$ then
            begin
                $cp_i := cp_i + 1$;
                COMPUTE$_i(cp_i, MSG_i)$;
                $MSG_i := \emptyset$;
                $MSG_i := MSG_i \cup queue_i(j)$ for all $n_j \in \mathcal{N}(n_i)$;
                $queue_i(j) :=$ nil for all $n_j \in \mathcal{N}(n_i)$
            end.
```

Synchronizer *Alpha* is only one of the possibilities. For generic computations like Algorithm *S_Alg*, there are two other synchronizers of interest. The first one is called synchronizer *Beta*, and requires for its operation a spanning tree already established on G, so the initial costs $Comm_0(Beta)$ and $Time_0(Beta)$ are no longer equal to zero, but depend instead on the distributed algorithm used to generate the tree. The processor at the tree's root has a special function in *Beta*, as it is responsible for gathering from all other processors the safety information needed to proceed to further pulses, and then broadcasting this information to all of them.

When a processor becomes safe with respect to a certain pulse and has received a *safe* message from "up-tree" (i.e., from the neighbors on the tree that stand between it and the tree's leaves), it then sends a *safe* message on its single tree channel that leads "down-tree" (i.e., in the direction of the root). The processor at the root, upon receiving *safe* messages on its adjacent tree channels, and being itself safe with respect to that pulse, sends a message "up-tree" indicating that the computation of a new pulse may be undertaken. Clearly, these control messages traverse only tree channels, so we have

$$Comm(Beta) = O(n)$$

and

$$Time(Beta) = O(n).$$

For generic computations, *Beta* does better than *Alpha* in terms of communication complexity, whereas *Alpha* is more efficient than *Beta* in terms of time complexity.

The other synchronizer of interest, called synchronizer *Gamma*, arises from a composition of synchronizers *Alpha* and *Beta*. In this composition, processors are conceptually grouped into clusters. Inside clusters, *Gamma* operates as *Alpha*; among clusters, it operates as *Beta*. The size and disposition of clusters are regulated by a parameter k such that $2 \le k \le n$ and in such a way that *Gamma*'s complexities are

$$Comm(Gamma) = O(kn)$$

and

$$Time(Gamma) = O\left(\frac{\log n}{\log k}\right).$$

As k varies, *Gamma* resembles more *Alpha* or *Beta*. Once again the costs of initialization $Comm_0(Gamma)$ and $Time_0(Gamma)$ are nonzero and depend on the mechanisms utilized.

4.2. SCHEDULING BY EDGE REVERSAL

4.2.1. Neighborhood constraints

In the study of parallel processing systems, we frequently encounter situations in which a group of agents request access to a shared resource in order to proceed with their particular computations. Such shared resources may be shared variables, files, or computing facilities of various sorts. A considerable amount of research has been dedicated in the past decades to the investigation of techniques that can be used to ensure the correctness of parallel computations on shared resources. Such investigations were at first directed toward programs executed by a single processor, as traditional operating systems, and then evolved into the study of distributed algorithms, such as the ones we have been considering in this book.

In order to illustrate the major problems arising when parallel processing agents utilize shared resources, let us consider the following problem, which has over the

years acquired a paradigmatic status in the field. The problem is known as the Dining Philosophers Problem (DPP), and is stated as follows. Five philosophers sit at a round table for dinner, with a fork placed on the table between the plates of each pair of adjacent philosophers (neighbors). It so happens that, in order to eat, every philosopher needs to use the two forks adjacent to his plate, which immediately means that two neighbors cannot eat simultaneously. DPP then asks for a protocol of communication among the philosophers to be devised that allows the philosophers to coordinate their eating in such a way that three basic constraints are not violated. First of all is the (already mentioned) ***mutual exclusion*** constraint, stating that no two neighbors can use the fork that they share simultaneously. Secondly, no ***deadlock*** is to be allowed, meaning that at all times at least one hungry philosopher must be able to eat (so such a rule as "whoever is hungry start by picking up the fork at the left" will not work). Thirdly, no ***starvation*** is to be allowed either, that is, every hungry philosopher must be able to eat within a "tolerable" amount of time.

DPP can be extended to a more general setting in which an arbitrary number of philosophers exist, and the "around-the-table" neighborhood relation is substituted with a more general binary relation in which any pair of philosophers may participate. Under the same rules for eating (only a little more bizarre), we say that each agent (philosopher) is under ***neighborhood constraints*** in order to carry on its computation (eat).

In this book, our interest in neighborhood-constrained systems comes from the need to simulate the PC automaton networks introduced in Chapter 1. In such networks, a set of nodes can have their states updated concurrently if and only if they constitute an independent set in G, i.e., a set that contains no neighbors. A node in a PC automaton network is then under neighborhood constraints to update its state.

4.2.2. The algorithm

In this section, we begin the study of a distributed algorithm to coordinate the operation of processors under neighborhood constraints. Our discussion will at first be developed under the assumption of a synchronous model of distributed computation. Whenever it is processor p_i's "turn" to compute, or in order to participate in the initialization phase, we assume that it does so by executing a procedure $\text{COMPUTE}_i(c, MSG')$. According to the notation in Section 3.1.5, here $c \ge 0$ identifies the clock pulse, and for $c > 0$ MSG' is the union of the various MSG's received by p_i along the pulses since its last "turn" to compute, and possibly of the set of initialization messages received from some of its neighbors. If $c = 0$, then $MSG' = \emptyset$, as this pulse corresponds to the initialization phase. We assume, mostly as we did in the case of Algorithm *Sas* given in Section 4.1.2, that in general COMPUTE_i sends exactly one message, denoted by *comp_msg*, to each of p_i's neighbors. If we further assume that neighbors take alternating "turns" to compute, then the set MSG' used by p_i contains, unless $c = 0$, exactly one *comp_msg* from each of its neighbors when it calls COMPUTE_i. The exception on the number of messages sent by COMPUTE_i happens for pulse $c = 0$, at which *comp_msg*'s may be sent as initialization messages to some neighbors only.

The distributed algorithm we employ is based on a mechanism whose functioning and properties rely heavily on graph-theoretic characteristics of G, which is, initially, assumed to have its edges oriented by an acyclic orientation. This acyclic orientation can be obtained in a distributed fashion by means of a quite straightforward procedure if the nodes can, as in our case, be assumed to have distinct identities. In this case, it suffices that edge (n_i, n_j) be oriented from n_i to n_j if and only if, say, $i < j$.

The following is then how the mechanism goes. At pulse $c = 1$, all sinks in the acyclic orientation of G constitute an independent set, and the corresponding processors may therefore compute. Note that at least one sink exists, as a consequence of the acyclicity of the orientation. When these processors finish computing, the direction of all edges incident to their nodes in G is reversed, so that these nodes become sources. At the beginning of pulse $c = 2$, a new orientation of G exists, and, if it contains at least one sink, then the corresponding processors may compute (Figure 4.1). The process is repeated indefinitely as long as the orientation resulting from the edge-reversal operations contains at least one sink. This process is known as *scheduling by edge reversal*, and, as we discuss in Section 4.2.3, is ensured to yield acyclic orientations at all pulses, provided the initial orientation is acyclic, and is in addition deadlock- and starvation-free. The latter two properties were explained in Section 4.2.1 in the context of DPP, and are also of great importance in the simulation of PC automaton networks.

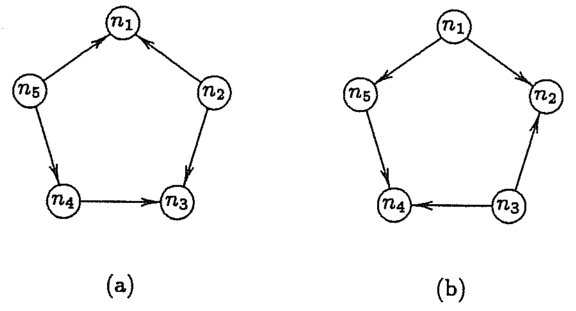


Figure 4.1. *The acyclic orientation in (a) has two sinks, n_1 and n_3. When the edges incident to these nodes are reversed, the acyclic orientation in (b) is obtained, in which n_1 and n_3 have become sources, and n_2 and n_4 are the new sinks.*

The reversal of edges in the scheduling by edge reversal mechanism is implemented by the sending of messages between processors. In fact, recalling that, for $c > 0$, when a processor computes it sends exactly one *comp_msg* to each of its neighbors, and in addition recognizing that neighbors do alternate in their "turns" to compute, we see that no special messages are needed to indicate edge reversal. When a processor p_i has received one *comp_msg* from each of its neighbors, this is an indication that n_i has become a sink and that it may then compute.

Although we described scheduling by edge reversal in a synchronous environment, its functioning in an asynchronous environment is equally simple and straightforward. In fact, the exact same mechanism works, without even requiring the use of a synchronizing technique as the ones described in Section 4.1. If, for a moment, we leave aside the initialization process alluded to earlier, then the asynchronous exchange of *comp_msg*'s, sent by processors when their nodes become sinks, is enough to ensure the correct functioning of scheduling by edge reversal in an asynchronous environment. Of course, the acyclic orientations and the independent sets of G can no longer be associated with specific points in time, but we can nonetheless identify global states in which the acyclic orientations and the corresponding independent sets appear.

We now present the details of the asynchronous version of scheduling by edge reversal, called Algorithm *Ser* (for *Scheduling by edge reversal*). Before we present its variables and input/action pairs, a little additional notation is needed. Given an acyclic orientation of G, and two neighbors n_i and n_j, we say that n_i is an *upstream neighbor* of n_j in that orientation if the edge (n_i, n_j) is oriented from n_i to n_j. Similarly, in this case n_j is said to be a *downstream neighbor* of n_i. Processors p_i and p_j shall be referred to likewise with respect to each other. Whenever we fail to mention the acyclic orientation, it should be understood that we refer to the "current" acyclic orientation. This is of course elusive in an asynchronous environment, but is precise if restricted to a node's neighborhood.

Algorithm *Ser* is started spontaneously by a group of processors P_1, which send *startup* messages to their neighbors. One such message, when received for the first time at a processor, triggers the sending of *startup*'s by that processor to its neighbors as well. Every processor, including those in P_1, only begin the initialization phase upon receiving a *startup* from all of its neighbors. A processor participates in this initialization phase by simply sending a *comp_msg* to every one of its downstream neighbors (processors corresponding to initial sinks do nothing then). These *comp_msg*'s sent in the initialization phase ensure that, when a node becomes a sink for the first time, the corresponding processor has received one *comp_msg* from each of its neighbors (this includes those nodes that are sinks initially, as they may be thought of as "becoming" sinks after the initialization phase).

The following are the variables maintained by p_i for Algorithm *Ser*. The number of times n_i has already been a sink is indicated by variable sc_i. The set MSG_i is used to store the various *comp_msg*'s for consumption when n_i next becomes a sink. A variable $go_i(j)$ for every neighbor p_j indicates whether a *startup* has already been received from p_j. A variable up_i is used to indicate whether p_i has already begun its participation in the computation, either spontaneously or in the wake of a *startup*. The number of neighbors from which a *comp_msg* is expected is kept in $pending_i$. Note, finally, that COMPUTE$_i$ does not depend, in the asynchronous environment, on a clock pulse number c. It is called, instead, as COMPUTE$_i(sc_i, MSG_i)$. The initialization phase is performed by COMPUTE$_i(0, \emptyset)$.

```
Algorithm Ser:
  ▷ Variables:
        $sc_i = 0$;
        $MSG_i = \emptyset$;
        $up_i$ = false;
        $go_i(j)$ = false for all $n_j \in \mathcal{N}(n_i)$;
        $pending_i = |\mathcal{N}(n_i)|$.
  ▷ Input:
        $msg$ = nil.
    Action at $p_i \in P_1$:
        $up_i$ := true;
        Send startup to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
  ▷ Input:
        $msg$ = startup.
    Action:
        $go_i(j)$ := true for $p_j = origin_i(msg)$;
        if not $up_i$ then
            begin
                $up_i$ := true;
                Send startup to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$
            end;
        if $go_i(j)$ for all $n_j \in \mathcal{N}(n_i)$ then
            COMPUTE$_i(sc_i, MSG_i)$.
  ▷ Input:
        $msg$ = comp_msg.
    Action:
        $MSG_i := MSG_i \cup \{msg\}$;
        $pending_i := pending_i - 1$;
        if $pending_i = 0$ then
            begin
                $sc_i := sc_i + 1$;
                COMPUTE$_i(sc_i, MSG_i)$;
                $MSG_i := \emptyset$;
                $pending_i := |\mathcal{N}(n_i)|$
            end.
```

Note that a processor, upon receiving a *comp_msg*, cannot tell whether that message was sent by a neighbor as part of the initialization phase or when its node became a sink. In either case, however, the processor is assured that, whenever the number of *comp_msg*'s received reaches a multiple of its number of neighbors, its node has become a sink.

4.2.3. Some properties

Scheduling by edge reversal possesses numerous interesting and important properties. In this section, we discuss the most relevant ones in the context of the simulation of PC automaton networks. Some details, as well as additional properties, are given in Appendices C and D. Our discussion here is developed under the assumption of a synchronous model of distributed computation (and so is the one in Appendix C). The extension of the most important ideas to an asynchronous model is simple, and can be found in the literature.

Throughout this section, and in the related portions of Appendices C and D, G is assumed to be connected and such that $n > 1$. If G is not connected, then the results we present are still valid for each of G's connected components. Note that these two assumptions are by no means too restrictive, as in their absence the neighborhood constraints discussed in Section 4.2.1 would hardly be meaningful.

As we saw in Section 4.2.2, scheduling by edge reversal is based on the evolution of orientations of G, starting at an initial acyclic orientation. After some initialization procedures at pulse $c = 0$, for $c > 0$ a succession of orientations is obtained (the initial acyclic orientation for $c = 1$) by turning all sinks into sources. As we remarked then, this succession is guaranteed to be endless if at least one sink can be found in each orientation. Clearly, in order to show that such is the case it suffices that we argue that all orientations are acyclic. Suppose, to the contrary, that an orientation with a directed cycle is obtained from an acyclic orientation. This must have involved the turning into a source of a node that is now on the cycle, as no other change in the acyclic orientation was performed but the turning of sinks into sources. However, no source can exist in a directed cycle, so all orientations must be acyclic if the first one is. As a by-product of this conclusion, we immediately have that the scheduling by edge reversal mechanism is deadlock-free, as in this context a situation of deadlock would be characterized by the inexistence of a sink in some orientation. As we show next, the mechanism is also starvation-free, where starvation, in this context, would be characterized by an indefinite number of pulses before a node became a sink.

Let $\omega_1, \omega_2, \ldots$ denote the sequence of acyclic orientations created by the edge-reversal process, and $\mathcal{J}_1, \mathcal{J}_2, \ldots$ denote the corresponding sets of sinks. For $n_i \in N$ and $k \geq 1$, let $m_i(k)$ be the number of times n_i appears in $\mathcal{J}_1, \ldots, \mathcal{J}_k$.

Theorem 4.1. *Consider two nodes $n_i, n_j \in N$ and let $r \geq 1$ be the number of edges on a shortest undirected path between n_i and n_j in G. Then $|m_i(k) - m_j(k)| \leq r$ for all $k \geq 1$.*

Proof: We use induction on the number of edges on a shortest undirected path between n_i and n_j. The case of one edge constitutes the basis of the induction, and then the assertion of the theorem holds trivially, as in this case n_i and n_j are neighbors in G, and must therefore appear in alternating sets in $\mathcal{J}_1, \mathcal{J}_2, \ldots$. As the induction hypothesis, assume the assertion of the theorem holds whenever a shortest undirected path between n_i and n_j has a number of edges no greater than $r - 1$. When n_i and n_j are separated by a shortest undirected path with r edges, consider any node n_ℓ (other than n_i and n_j) on this path and let d be the number of edges between n_i and n_ℓ on the path. By the induction hypothesis,

$$|m_i(k) - m_\ell(k)| \leq d$$

and

$$|m_\ell(k) - m_j(k)| \leq r - d,$$

yielding

$$|m_i(k) - m_j(k)| \leq r,$$

thence the theorem. ■

Theorem 4.1 is in fact more than a mere starvation-freedom statement, as not only does it imply that all nodes become sinks within a bounded number of pulses, but it also establishes a bound on the relative frequency with which nodes become sinks.

The number of distinct acyclic orientations of G is of course finite, so the sequence $\omega_1, \omega_2, \ldots$ must at some pulse embark in a periodic repetition of orientations, which we call a *period of orientations*, or simply a *period.* Orientations in a period are said to be *periodic orientations* (Figure 4.2). The next corollary establishes an important property of periods.

Corollary 4.2. *The number of times that a node becomes a sink in a period is the same for all nodes.*

Proof: Suppose, to the contrary, that two nodes n_i and n_j exist that become sinks different numbers of times in a period. Suppose, in addition, that a shortest undirected path between n_i and n_j in G has r edges. Letting p be the number of orientations in the period and $k = (r+1)p$ yields

$$|m_i(k) - m_j(k)| \geq r + 1,$$

which contradicts Theorem 4.1. ■

Clearly, the period is unequivocally determined given ω_1. We let $m(\omega_1)$ denote the number of times that nodes become sinks in this period, and $p(\omega_1)$ denote the number of orientations in the same period.

One issue of great interest is the "amount of concurrency" that scheduling by edge reversal is capable to yield from an initial acyclic orientation ω_1. The importance of this issue comes from the very nature of the neighborhood constraints that motivated this entire section, and from the intuitive realization that the choice of ω_1 greatly influences the number of nodes that become sinks concurrently (Figures 4.2 and 4.3).

We propose to measure the concurrency attainable from ω_1, denoted by $Conc(\omega_1)$, as an average, over a large number of pulses and over n, the number of nodes, of the number of times each node becomes a sink in those pulses. More formally,

$$Conc(\omega_1) = \lim_{k\to\infty} \frac{1}{kn} \sum_{n_i \in N} m_i(k).$$

Clearly, we get more concurrency as more nodes become sinks earlier.

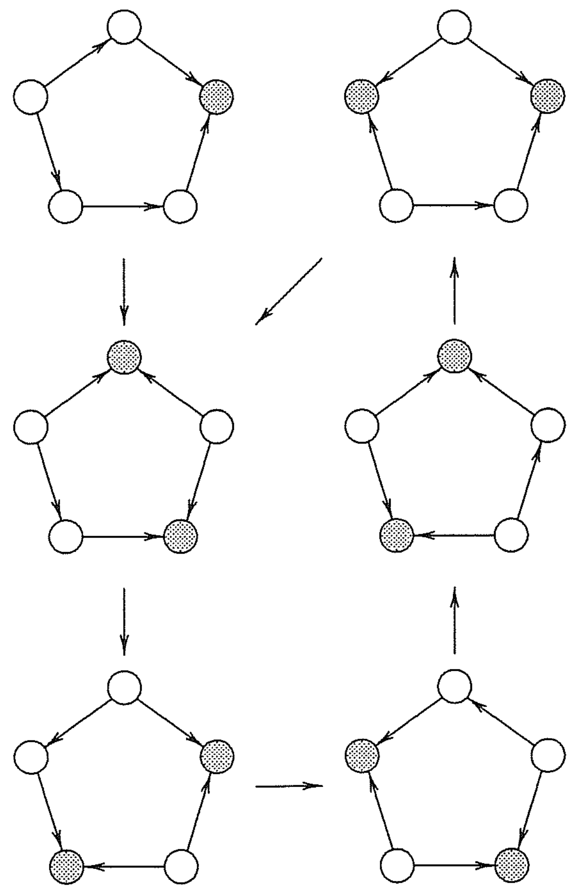


Figure 4.2. *A period of five orientations results from the edge-reversal mechanism started at the orientation shown in the upper left corner of the figure, which is outside the period. In this period, every node becomes a sink twice. Under different starting conditions (i.e., a different initial orientation), this number might be different, possibly yielding less concurrency (cf. Figure 4.3).*

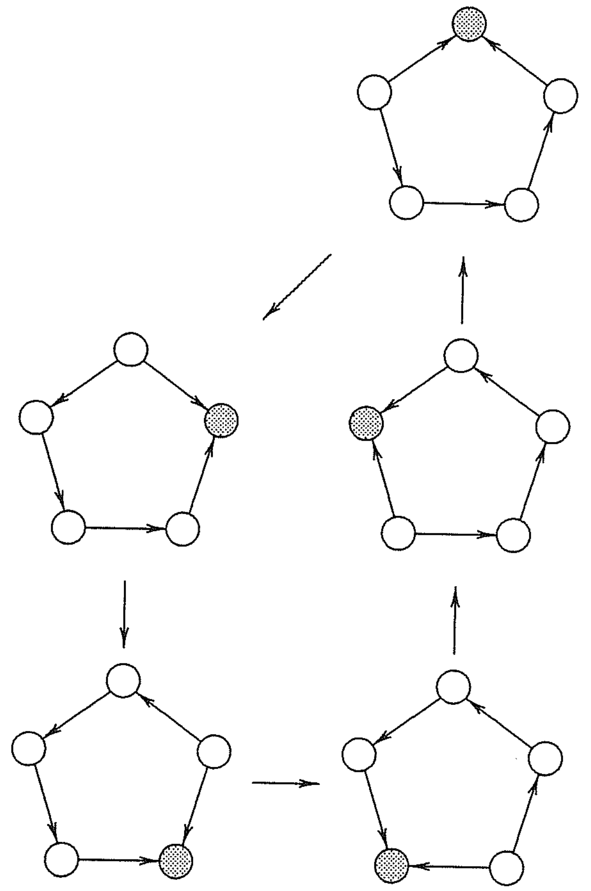


Figure 4.3. *This figure shows a period of five orientations that can only result from starting the edge-reversal mechanism at one of its own orientations. It is simple to see that no other orientation exists from which this period can result, as none of the period's orientations has more than one source that is adjacent to all of its sinks. In this period, every node becomes a sink only once, which should be contrasted with the situation shown in Figure 4.2. Less concurrency is then in this case obtained in comparison with that figure.*

Theorem 4.3. *$Conc(\omega_1) = m(\omega_1)/p(\omega_1)$.*

Proof: For some $\ell \geq 1$, let ω_ℓ be the first periodic orientation in $\omega_1, \omega_2, \ldots$. For $k \geq \ell$, the first k orientations of $\omega_1, \omega_2, \ldots$ include $\lfloor (k - \ell + 1)/p(\omega_1) \rfloor$ repetitions of the period, so

$$k = \left\lfloor \frac{k - \ell + 1}{p(\omega_1)} \right\rfloor p(\omega_1) + u$$

and

$$\sum_{n_i \in N} m_i(k) = n \left\lfloor \frac{k - \ell + 1}{p(\omega_1)} \right\rfloor m(\omega_1) + v,$$

where $\ell - 1 \leq u \leq \ell + p(\omega_1) - 2$ and $u \leq v \leq nu$. The theorem then follows easily in the limit as $k \to \infty$. ■

It follows immediately from Theorem 4.3 that

$$\frac{1}{n} \leq Conc(\omega_1) \leq \frac{1}{2}.$$

This is so because it takes at most n pulses for a node to become a sink (the longest directed distance to a sink is $n - 1$), so

$$m(\omega_1) \geq \frac{p(\omega_1)}{n},$$

and because the most frequently that a node can become a sink is in every other pulse, so

$$m(\omega_1) \leq \frac{p(\omega_1)}{2}.$$

If G is a tree, then it can be argued relatively simply that $Conc(\omega_1) = 1/2$, regardless of the initial orientation ω_1. If G is not a tree, then, interestingly, $Conc(\omega_1)$ can also be expressed in purely graph-theoretic terms, without recourse to the dynamics of the edge-reversal scheduling mechanism. For such, let κ denote an undirected cycle in G. Let also $n^+(\kappa, \omega_1)$ and $n^-(\kappa, \omega_1)$ denote the number of edges in κ oriented by ω_1 clockwise and counter-clockwise, respectively. Define

$$\rho(\kappa, \omega_1) = \frac{1}{\kappa} \min\{n^+(\kappa, \omega_1), n^-(\kappa, \omega_1)\},$$

and let K denote the set of all of G's undirected cycles.

Theorem 4.4. *If G is not a tree, then $Conc(\omega_1) = \min_{\kappa \in K} \rho(\kappa, \omega_1)$.*

Proof: See Appendix C. ■

There are in the literature additional results concerning scheduling by edge reversal that we do not explicitly reproduce here. Some are positive, as the one that states that this mechanism is optimal (provides most concurrency) among all schemes that operate under neighborhood constraints and require neighbors to have alternating "turns." Other results are negative, as for example the computational intractability of finding the initial acyclic orientation ω_1 that optimizes $Conc(\omega_1)$.

4.3. TEMPLATE ALGORITHMS

4.3.1. A generic architecture

In this section and in the remainder of Section 4.3, we describe template algorithms for the distributed parallel simulation of the automaton networks we consider in Chapters 5 through 10. These template algorithms provide the general appearance of the simulation algorithms, and involve calls to procedures whose functioning depends on the particular class of automaton networks under consideration. The actual contents of such procedures are specified in the appropriate chapters.

As we argue in Sections 4.3.2 and 4.3.3, the most natural approach to the distributed parallel simulation of FC and PC automaton networks is to employ Algorithms *Sas* and *Ser*, respectively. These algorithms appear in Sections 4.1.2 (Algorithm *Sas*) and 4.2.2 (Algorithm *Ser*), and have been given without any concern to the issue of how they are to terminate. This is then one major issue in the discussion of a generic architecture for our distributed parallel simulator.

In our discussion in Section 3.3, we indicated essentially two approaches to termination detection in distributed computations. The first approach, applicable to distributed computations in general, is based on the notion of global states of the computation (discussed in Section 3.2), and consists of a repetitive evaluation of global states until one is found in which the computation has terminated. The second approach, applicable only within the so-called class of diffusing computations, is based on the existence of an additional agent (we called it processor p_0) whose function is to serve as the sole initiator of the computation and to detect the computation's termination eventually.

As we shall see in the remainder of Section 4.3, in most cases of interest in this book the approach that checks global states for termination is the most adequate, and it is based on this approach that our termination-detection strategy will be developed. In fact, what we need in our simulators are strategies for detecting the end of the simulation, and this is where the checking of global states comes in. Once the end of the simulation has been detected, the computation's termination can be "provoked." The approach that operates on diffusing computations can also be of great importance, but is only applicable to more advanced versions of the simulators we describe in this book. Such versions are, however, still the subject of research, and depend on the new results on synchronizers that are beginning to appear in the literature.

The architecture we utilize for our simulators is conceptually very simple. Besides the n processors $p_1, \ldots, p_n$, interconnected to one another according to the edges of G, we also employ an additional processor, called p_0, directly interconnected to all the other n processors (Figure 4.4). Processor p_0 has some functions that are common to all simulators, as the "waking up" of the processors in the set P_1 of spontaneous initiators, and the maintenance of logs and computation of statistics along the simulation. In some simulators, p_0 is also responsible for detecting the end of the simulation, and then broadcasting a termination order to all processors. As we discuss later, simulators in which p_0 does not have this additional function are such that the termination process is so simple that it can be carried out practically at no cost by the n processors themselves.

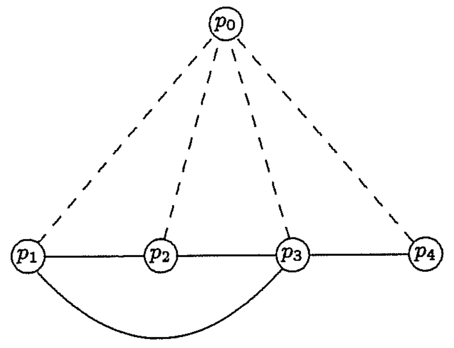


Figure 4.4. *Besides the n processors (four, in this case), one for each node in G, an additional processor, p_0, is employed in this generic architecture for our distributed parallel simulators of FC and PC automaton networks. Processor p_0 performs the functions of initiating the other processors' computation and is sometimes also responsible for detecting the simulation's termination. It also performs additional administrative functions.*

4.3.2. Fully concurrent automaton networks

We know from Chapter 1 that for FC automaton networks the updating rule says that the states of all nodes are to be updated at all pulses $s > 0$, starting with the nodes' states at $s = 0$. This behavior is summarized in (1.2).

The simulation of an FC automaton network consists of producing, for all $s \geq 0$, the states $x_i(s)$ of all nodes $n_i \in N$. Given the nature of its updating rule, an FC automaton network is then very naturally simulated by Algorithm *Sas*, introduced in Section 4.1.2. In this algorithm, the procedure $\text{COMPUTE}_i(cp_i, MSG_i)$, where cp_i is the pulse number and MSG_i either is an empty set (if $cp_i = 0$) or contains exactly one message from each of p_i's neighbors, can be encoded as the updating function of the automaton network with $cp_i = s$. In this case, MSG_i contains, for $s > 0$, information concerning n_i's neighbors' states at $s - 1$. For the FC automaton networks we describe in Chapters 5 through 7, such information will be the very states of the neighbors. The FC automaton networks we discuss in this book are cellular automata (Chapter 5), analog Hopfield neural networks (Chapter 6), and other types of analog neural networks (Chapter 7).

The simulation is started by p_0 by sending a "wake-up" message to each of the processors in P_1, i.e., the processors that are in Algorithm *Sas* assumed to start the computation spontaneously. As we are utilizing the additional processor p_0, we assume that the procedure COMPUTE$_i$ also sends one message to p_0 at all pulses containing the state of n_i at that pulse. With this information, p_0 can then detect the end of the simulation and then forward a termination order to each of the n processors. This is in general the case with the three classes of FC automaton networks we study in this book, for which the end of the simulation is detected when convergence to a fixed point occurs. For cellular automata, particularly, it may also sometimes be the case that the simulation is to be terminated simply after a certain number of updates has been performed per node, in which case p_0 no longer performs the function of convergence detector, but can still be used to maintain logs and compute statistics. We shall discuss this question in detail when we come to the chapter that treats each model individually.

When used to detect the end of the simulation, p_0 must maintain a two-dimensional array with n columns whose rows correspond to each of the pulses $s \geq 0$ and contain the states received from the processors at each of the pulses (Figure 4.5). From our study in Section 3.2, it should be clear that each such row corresponds to a global state of the computation carried out by the n processors. The end of the simulation is then detected by p_0 by comparing the states of the nodes in two successive global states (two successive rows of the array), and then checking them for convergence. It should also be clear, from our discussion in Section 4.1, that, although reports from the processors may arrive at p_0 in such a way that a row is not completely filled before information arrives for the next row, room for no more than n rows is necessary.

Let *stop* be the message used by p_0 to terminate the computation when it detects the end of the simulation. This message is sent to each of the n processors. Upon receiving a *stop*, a processor forwards it to all of its neighbors, and only terminates upon receiving similar messages from each of them. This implies the following modifications in the structure of Algorithm *Sas*.

- An additional input/action pair to treat the reception of *stop* by p_i. The action in this case is to set up_i = **false** and to send *stop* to every p_j such that $n_j \in \mathcal{N}(n_i)$ for the first time, and to simply decrement a counter of *stop*'s received for the other times.

- The input/action pair triggered by the reception of a *comp_msg* is to be considered only when up_i = **true**. This allows *comp_msg*'s received after a *stop* to be ignored.

The following is then the operation of COMPUTE$_i$ for the simulation of FC automaton networks.

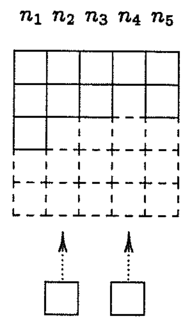


Figure 4.5. *Processor p_0 maintains an $n \times n$ (in this case, 5×5) array where information originating from the other processors is stored. There is one column for each processor (respectively, for each node in G). When new information arrives from a processor, it is placed at the topmost empty position of the corresponding column. Whenever a row becomes full, a global state of the distributed computation happening on the n processors is completed.*

```
COMPUTE_i($cp_i$, $MSG_i$):
    if $cp_i = 0$ then
        INITIALIZE_i
    else
        begin
            UPDATE_i($MSG_i$);
            NOTIFY_i
        end.
```

In Chapters 5 through 7, we present procedures INITIALIZE$_i$, UPDATE$_i$, and NOTIFY$_i$ for the corresponding classes of FC automaton networks. Procedures INITIALIZE$_i$ and NOTIFY$_i$ perform communication-related functions by sending n_i's initial and updated states, respectively, wherever they are to be sent. Procedure UPDATE$_i$ implements the updating function for n_i.

4.3.3. Partially concurrent automaton networks

For PC automaton networks, the updating rule we saw in Chapter 1 is such that, at pulse $s > 0$, only the nodes in the independent set $\mathcal{I}_s$ are to be updated, starting with the nodes' states at $s = 0$. The only requirement concerning the sequence of independent sets $\mathcal{I}_1, \mathcal{I}_2, \ldots$ is that every node appear infinitely often in it. (1.3) summarizes this behavior.

The simulation of a PC automaton network consists of producing, for all $s \geq 0$, the states $x_i(s)$ of all nodes $n_i \in \mathcal{I}_s$. The nature of a PC automaton network's updating rule and the properties of the edge-reversal mechanism discussed in Section 4.2 indicate that such a network is very naturally simulated by Algorithm *Ser*, given in Section 4.2.2. By Theorem 4.1, every node appears infinitely often in the sequence of independent sets $\mathcal{J}_1, \mathcal{J}_2, \ldots$ generated by edge reversal, so the simulation of PC automaton networks can be carried out by setting $\mathcal{I}_s = \mathcal{J}_s$ for all $s > 0$. In Algorithm *Ser*, the procedure $\text{COMPUTE}_i(sc_i, MSG_i)$ can be encoded as the updating function of the automaton network with $sc_i \leq s$ if $n_i \in \mathcal{I}_s$. Here, sc_i indicates the number of times n_i has been a sink (has appeared in the sequence $\mathcal{J}_1, \mathcal{J}_2, \ldots$) and MSG_i contains exactly one message from each of p_i's neighbors (except for $sc_i = 0$, in which case it is an empty set). MSG_i then contains, for $s > 0$ such that $n_i \in \mathcal{I}_s$, information concerning n_i's neighbors at $s - 1$. For the PC automaton networks we describe in Chapters 8 through 10, this information is in general the node's neighbors' states, except in the case of Bayesian networks, discussed in Chapter 10. The other two classes of PC automaton networks discussed in this book are binary Hopfield neural networks (Chapter 8) and Markov random fields (Chapter 9).

As in the case of FC automaton networks discussed in Section 4.3.2, p_0 is responsible for starting the computation by sending a "wake-up" message to each of the processors in the set P_1 of spontaneous initiators. Here we also assume that COMPUTE_i sends a message to p_0 as well, initially and whenever n_i becomes a sink. This message contains n_i's state in that occasion. This information can be used by p_0 either to detect the end of the simulation or for logging and statistical purposes.

In the case of PC automaton networks, however, the termination process will only be conducted by p_0 in the case of binary Hopfield neural networks. In this case, then, the same $n \times n$ array used in Section 4.3.2 for p_0 to store global states for further comparison is also needed (Figure 4.5). This is justified by an immediate application of Theorem 4.1, which states bounds on how far processors may drift apart from one another concerning the number of times their nodes become sinks in the edge-reversal scheduling mechanism. When p_0 does detect the end of the simulation, it then sends a *stop* message to all processors, which in turn proceed to flushing their incident channels of messages by means of the exchange of *stop* messages with their neighbors. The process here is exactly the same as we discussed in Section 4.3.2, and entails the same modifications to Algorithm *Ser* as were needed to Algorithm *Sas*.

The simulation of the other two classes of PC automaton networks, Markov random fields and Bayesian networks, is performed in such a way that each node is updated a pre-defined number of times, the same for all nodes. In this case, the intervention of p_0 in the termination process is no longer needed, as each processor simply has to update its node until the number of updates expires. Given the properties of edge reversal studied in Section 4.2 (particularly Theorem 4.1), it is simple to understand that all processors will make it to updating their nodes the same pre-specified number of times. If K is this number, then let p_i be the first processor to update its node K times (we make these comments under the same assumption of a synchronous model we used in Section 4.2.3 for the study of edge-reversal properties). If this is the case with more than one processor, they are

certainly not neighbors. By Theorem 4.1, p_i's neighbors have updated their nodes exactly $K-1$ times, some of its neighbors' neighbors have updated their nodes $K-2$ times, and so on. The current orientation of G's edges is from the nodes that have been more updated to the ones that have been less updated. The process will then continue until the original acyclic orientation is restored, at which time all nodes will have been updated K times. Processors that have performed K updates may stop, but must nevertheless be on the lookout for messages from their neighbors that have not stopped yet. Specifically, exactly one such message is expected to arrive at a processor from each neighbor that was originally upstream from it. This argument is developed more formally in Appendix D.

When termination is not detected by p_0, the following change to Algorithm *Ser* must be made.

- After p_i has updated n_i the pre-specified number of times, up_i is set to **false**. The input/action pair triggered by the reception of a *comp_msg* is then to be considered only when $up_i =$ **true**.

The following is then the operation of COMPUTE_i for the simulation of PC automaton networks.

```
COMPUTE_i($sc_i$, $MSG_i$):
    if $sc_i = 0$ then
        INITIALIZE_i
    else
        begin
            UPDATE_i($MSG_i$);
            NOTIFY_i
        end.
```

In Chapters 8 through 10, we present procedures INITIALIZE_i, UPDATE_i, and NOTIFY_i for the corresponding classes of PC automaton networks. Procedures INITIALIZE_i and NOTIFY_i perform communication-related functions by sending n_i's initial and updated states (or other information, in the case of Bayesian networks in Chapter 10), respectively, wherever they are to be sent. Procedure UPDATE_i implements the updating function for n_i.

4.4. BIBLIOGRAPHIC NOTES

The synchronizers described in Section 4.1 are based on Awerbuch (1985a, 1985b). Algorithms to build the spanning tree needed by synchronizer *Beta* can be found in Gallager, Humblet, and Spira (1983), Awerbuch (1987), and Chin and Ting (1990). The research on algorithm synchronizers is still progressing, and further results can affect the algorithms presented in this book. For example, techniques similar to the ones introduced by Awerbuch and Peleg (1990) can be used to reduce the communication complexity of simulating FC automaton networks.

The issue of resource sharing discussed in Section 4.2 can be found in books on operating systems and concurrent programming, such as Ben-Ari (1982), Peterson and Silberschatz (1985), Maekawa, Oldehoeft, and Oldehoeft (1987), and Andrews (1991). Additional references on the subject are Lynch (1980, 1981) and Andrews and Schneider (1983). DPP was introduced by Dijkstra (1968), and later received considerable attention, as can be seen in the aforementioned books and in Chang (1980) and Rabin and Lehmann (1981). Chandy and Misra (1984) present the version of DPP in which the neighborhood structure is generalized.

The scheduling by edge reversal mechanism and its properties (including optimality and intractability) are from Barbosa and Gafni (1987, 1989b), where the consequences of an asynchronous model are also addressed. The mechanism is also treated by Bertsekas and Tsitsiklis (1989). The same mechanism had already been utilized in other contexts, as by Gafni and Bertsekas (1981) for the routing of packets in computer networks, and by Chandy and Misra (1984) in the context of DPP. Part of the analysis of edge reversal given in Barbosa and Gafni (1987, 1989b) relies on the notion of node multicolorings in graphs, treated by Stahl (1976).

The template algorithms given in Section 4.3 for the distributed parallel simulation of FC and PC automaton networks are based on the algorithms given by Barbosa and Lima (1990) for the simulation of Hopfield neural networks. As we already mentioned, synchronizer developments in the spirit of those reported by Awerbuch and Peleg (1990) can have a significant impact on the template for the simulation of FC automaton networks.

Part 3

Fully concurrent automaton networks

FC automaton networks are the subject of the chapters in this part. The overall approach is to present various FC automaton network models, along with their main properties, and to discuss the most relevant application areas of each model. Algorithms for the sequential and distributed parallel simulation of each model are also provided.

Part 3 comprises Chapters 5 through 7. Chapter 5 is devoted to cellular automata, Chapter 6 to analog Hopfield neural networks, and Chapter 7 to other analog neural networks of importance in the solution of mathematical problems as systems of linear algebraic equations and linear programs.

5

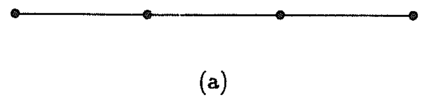
Cellular automata

This chapter is devoted to a discussion of cellular automata, which are introduced in Section 5.1 along with their main properties. Algorithms for the simulation of cellular automata are given in Section 5.2 (the distributed parallel algorithm is based on the characterization of cellular automata as FC automaton networks). Other properties of interest related to one-dimensional cellular automata are discussed in Section 5.3, while in Section 5.4 a brief discussion on probabilistic cellular automata is presented. Bibliographic notes are given in Section 5.5.

5.1. THE MODEL AND BASIC PROPERTIES

Consider a set S and a symmetric binary relation R on S. The 1-*transitive closure* of R, denoted by R^1, is defined to be a binary relation equal to R. For some integer $r > 1$, the r-*transitive closure* of R, denoted by R^r, is obtained from R^{r-1} by adding to R^{r-1} every pair $(s_i, s_j) \in S \times S$ such that, for some $s_k \in S$, $(s_i, s_k) \in R^{r-1}$ and $(s_k, s_j) \in R$. An FC automaton network for which G has as edge set the r-transitive closure of the edge set of a multidimensional lattice (an undirected graph that can be "drawn" as a lattice of multiple dimensions), for some $r > 0$, is known as a *cellular automaton*. We shall refer to this multidimensional lattice as the *underlying lattice* of the cellular automaton (Figure 5.1). Clearly, for $r = 1$ the underlying lattice and G are the same undirected graph.

The underlying lattice of a cellular automaton may have several variations of interest, depending on how its boundaries are defined in the various dimensions. Normally, every node in N has in a δ-dimensional lattice, for $\delta \geq 0$, 2δ neighbors. Of these 2δ neighbors, exactly two have positions in the lattice that differ from the position of the node in each of the δ dimensions. An exception to this rule occurs at the boundaries, where nodes may have as few as δ neighbors in the lattice, and sometimes it is interesting to consider the so-called *periodic boundary conditions* for one or more of the dimensions in which the underlying lattice does not stretch infinitely. Periodic boundary conditions for a given dimension are characterized by



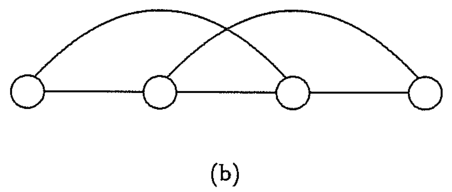


Figure 5.1. *The four-node lattice shown in part (a) is the underlying lattice of the cellular automaton for which G is shown in part (b). In this case, $r = 2$.*

treating as neighbors in the lattice the extreme nodes in that dimension, that is, the nodes that otherwise have one single neighbor in that dimension. A multidimensional lattice with periodic boundary conditions in a dimension is said to be *cylindrical* with respect to that dimension; the same denomination is inherited by the corresponding cellular automaton (Figure 5.2).

The notions of the *state* of a node in a cellular automaton and of the *state* of a cellular automaton are, in view of the automaton's definition, the same as the corresponding notions in the more general context of automaton networks discussed in Chapter 1. The state $x_i(s)$ of every node $n_i \in N$ at time $s \geq 0$ is a member of the finite set $D = \{0, 1, \ldots, d-1\}$ for some $d > 0$. The updating function f is then a function which for $s > 0$ assigns one of the values in D to the state of n_i, and utilizes for such the states of n_i and of all of n_i's neighbors in G (i.e., the nodes in $\mathcal{N}(n_i)$) at time $s - 1$. Because the edge set of G is by definition the r-transitive closure of the edge set of a multidimensional lattice for some $r > 0$, the updating function may be regarded as taking into account, when updating n_i's state, the influence of all nodes that lie in the lattice within a distance r of n_i. The value of r is then referred to as the updating function's *radius*.

Cellular automata are in general *deterministic* models, meaning that at any state the updating function f can only produce exactly one of the automaton's d^n possible states as its next state. If, in addition to being deterministic, the cellular automaton is also *finite* (this is the case in which N is finite), then the sequence of states generated by f from any initial state does necessarily become periodic at some time $s \geq 0$. The set of states that are repeated in this inevitable periodic behavior is a *limit cycle* of the automaton, and the number of states in a limit cycle

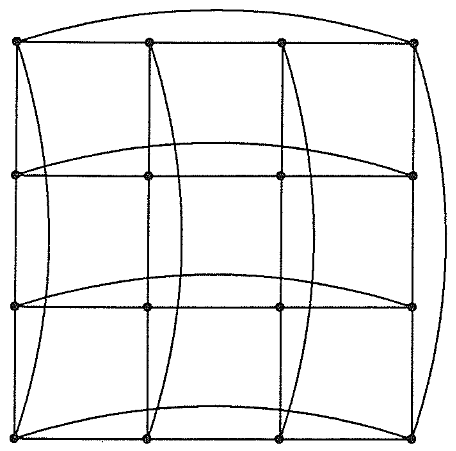


Figure 5.2. *This two-dimensional lattice has periodic boundary conditions for both dimensions, and is then cylindrical with respect to both of them.*

is the cycle's *length*. If the length of a limit cycle is equal to one, then it is called a *fixed point*. An analysis of such periodic behavior for some simple cellular automata is discussed in Section 5.3.

If the node set N is infinite (and then the cellular automaton is said to be *infinite*), or if the updating function f is nondeterministic (and then the cellular automaton is said to be *nondeterministic*), then no periodic behavior is guaranteed to occur at any time $s \geq 0$. This is so because the number of different states of an infinite cellular automaton is of course infinite, and because in the nondeterministic case the updating function f may (at least at some of the states) choose among more than one possible next state. Some simple cellular automata whose updating functions embody probabilistic decisions are discussed in Section 5.4. Cellular automata treated everywhere else in this chapter are assumed to be deterministic.

Beyond this first classification of cellular automata as finite, infinite, deterministic, and nondeterministic, studies have been conducted to attempt a classification based on the behavior of the automata on every possible starting state (or at least on most of the possible starting states). One of these studies has pointed at very interesting possibilities, and has culminated with the conjecture that all one-dimensional cellular automata fall into one of the following four categories.

(i) Evolution leads to a homogeneous state, i.e., a state in which all nodes have the same state.

(ii) Evolution leads to a state or states comprising, respectively, a set of stable or periodic structures which are simple and separated from one another.

(iii) Evolution leads to states characterizing a chaotic pattern.

(iv) Evolution leads to states comprising structures which are complex and localized, sometimes "long-lived."

These categories provide a qualitative description of the behavior of one-dimensional cellular automata. An automaton is said to belong to one of the categories if for "almost all" initial states its behavior endows it with the properties of that category. Albeit the informality of this categorization, some interesting consequences can be observed. Suppose, for example, that the updating function f of an infinite cellular automaton has one single argument given by the sum of the states of all nodes within its radius, i.e., it is such that

$$x_i(s) = f\left(\sum_{n_j \in \{n_i\} \cup \mathcal{N}(n_i)} x_j(s-1)\right) \tag{5.1}$$

for all $n_i \in N$ and all $s > 0$. Then for $d = 2$ and $r = 1$ indications have been found that half of the possibilities for f represent automata in category (i), while categories (ii) and (iii) receive one quarter of the functions each and category (iv) is not represented. If r is increased as d is kept constant, then apparently category (iii) becomes by far the most common, while the population in categories (i) and (ii) dwindle and that in category (iv) increases slightly. When d is increased, this same behavior is also observed, although category (iv) tends to become a little less rare (but always comparatively rare, nonetheless).

For examples of one-dimensional cellular automata in categories (i) through (iv) with $r = 2$ and $d = 2$ (i.e., node states may depend on the states of nodes located as far as two hops away in the lattice and are either 0 or 1), consider the updating functions f_1, f_2, f_3, and f_4 given by

$$f_1(y) = \begin{cases} 1, & \text{if } y = 2; \\ 0, & \text{otherwise;} \end{cases} \tag{5.2}$$

$$f_2(y) = \begin{cases} 1, & \text{if } y = 3 \text{ or } y = 4; \\ 0, & \text{otherwise;} \end{cases} \tag{5.3}$$

$$f_3(y) = \begin{cases} 1, & \text{if } y = 2 \text{ or } y = 3; \\ 0, & \text{otherwise;} \end{cases} \tag{5.4}$$

$$f_4(y) = \begin{cases} 1, & \text{if } y = 2 \text{ or } y = 4; \\ 0, & \text{otherwise,} \end{cases} \tag{5.5}$$

where y represents the summation that appears as argument of f in (5.1). The behavior of infinite cellular automata following the updating functions of (5.2) through (5.5) falls, respectively, into categories (i) through (iv) for "almost all" initial states (Figures 5.3(a) through 5.3(d)).

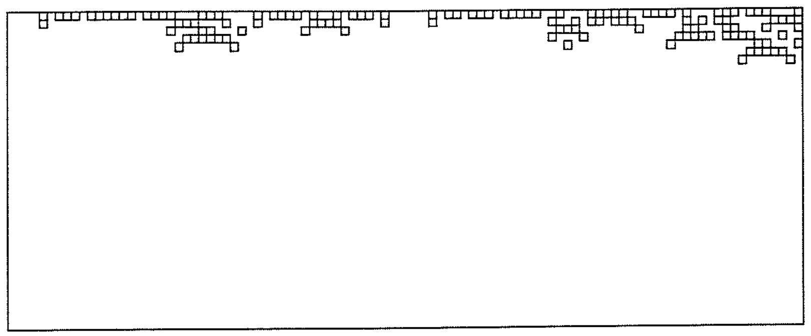


Figure 5.3(a). *The outer rectangle in this figure is to be regarded as an array having one row for each of forty pulses and one column for each of 100 nodes of an infinite one-dimensional cellular automaton with $r = 2$ and $d = 2$. The topmost row corresponds to pulse $s = 0$. In the case shown in this figure, the evolution according to (5.2) is depicted, employing squares to indicate states equal to 1 and blank spaces to indicate otherwise. The initial state was generated randomly. After a few pulses, the cellular automaton enters a homogeneous state, in this case one in which all nodes have states equal to 0. This is characteristic of an automaton belonging to category (i).*

5.2. SIMULATION ALGORITHMS

5.2.1. The sequential algorithm

The sequential simulation of a cellular automaton can be achieved in a rather simple way, as such an automaton is by definition an instance of FC automaton networks. An algorithm with this end, called Algorithm *Seq_Cellular_Automaton*, is given next. Starting with the initial states $x_i(0)$, the algorithm generates $x_i(s)$ for all $s > 0$ and all $n_i \in N$, and is based on the updating rule given in (1.2). As we discussed in Section 5.1, the limiting behavior of cellular automata for large values of s depends rather intimately on the particular automaton at hand, although in general two broad categories of long-term behavior can be expected. Cellular automata in the first category are finite and deterministic, and then evolve into limit cycles or fixed points, whereas those in the second category do not exhibit any such periodic behavior, as a possible consequence of being infinite or nondeterministic. The simulation of a cellular automaton in the former category can be terminated when the periodic behavior is entered, whereas to terminate the simulation of a (finite) cellular automaton in the latter category there does not appear to be any consistent criterion but to simply run it for a pre-specified number of updates per

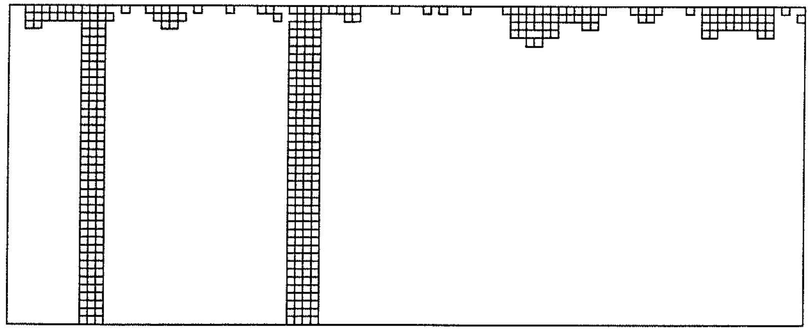


Figure 5.3(b). *This figure is like Figure 5.3(a), except that (5.3) has been employed to guide the automaton's evolution, which tends to a set of simple and separated periodic structures. This characterizes a category-(ii) behavior.*

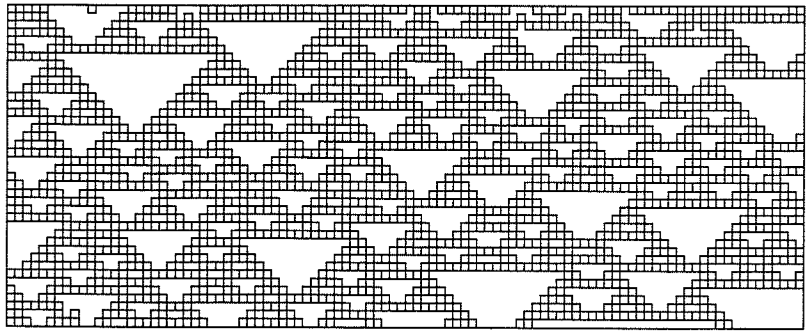


Figure 5.3(c). *This figure is like Figure 5.3(a), except that (5.4) has been employed as the automaton's updating function. A chaotic pattern emerges as a consequence, thereby characterizing a category-(iii) behavior.*

node, or, equivalently, until a maximum value of s is reached, say K. These are the strategies we employ in Algorithm *Seq_Cellular_Automaton*. In the algorithm, the detection of periodicity is referred to as the automaton's having been "trapped."

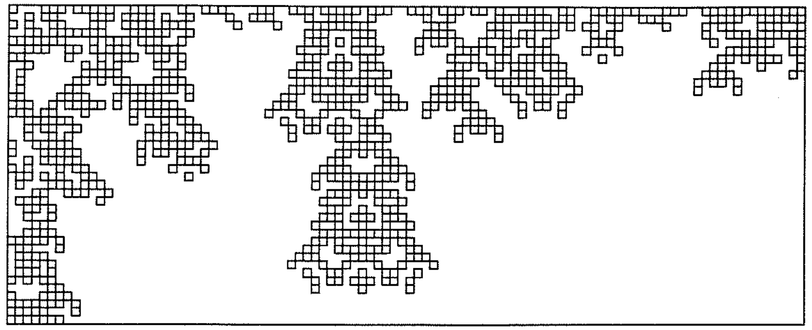


Figure 5.3(d). *This figure is like Figure 5.3(a), except that (5.5) has been employed as the updating function of the cellular automaton. Localized, complex structures appear which last for quite a few pulses. This characterizes a category-(iv) behavior.*

Algorithm *Seq_Cellular_Automaton:*

```
$s := 1$;
**repeat**
    **for** $i := 1$ **to** $n$ **do**
        $x_i(s) = f(x_j(s-1);\ n_j \in \{n_i\} \cup \mathcal{N}(n_i))$;
    $s := s + 1$
**until** "trapped" **or** $s > K$.
```

Depending on the particular cellular automaton under study, the detection of periodicity may or may not be simple to handle in Algorithm *Seq_Cellular_Automaton*, as the number of states in the period may or may not be known beforehand. We shall in Section 5.3 discuss some simple cases in which some analysis of the eventual periodic behavior can be done.

5.2.2. The distributed parallel algorithm

Cellular automata are by definition instances of the class of FC automaton networks, and as such can be simulated by a distributed parallel algorithm that follows the template given in Section 4.3.2. Depending on the particular cellular automaton under consideration, processor p_0 may or may not have an active participation as a detector of the simulation's termination. This is so because, as we remarked in Section 5.2.1, it may happen that for large values of s the automaton enters a periodic behavior, as it may as well happen that no characteristics of periodicity appear to be present in the long-run behavior of the automaton. While in the former case the simulation may terminate when the periodic behavior is detected (this can be done by p_0), in the latter case the simulation is run for a pre-specified number K of updates per node.

In order to obtain a distributed parallel simulator of cellular automata, we must specify the procedures of the template in Section 4.3.2, namely INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for every processor p_i such that $n_i \in N$. The set MSG_i contains, as we know, exactly one message from each of p_i's neighbors, in this case the state of the corresponding node for use in the updating of n_i's state. The three procedures are given next, and are, as Algorithm *Seq_Cellular_Automaton* of Section 5.2.1, based on (1.2).

```
INITIALIZE_i:
    1. Let $x_i := x_i(0)$;
    2. Send $x_i$ to $p_0$;
    3. Send $x_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
UPDATE_i($MSG_i$):
    1. Let $x_j \in MSG_i$ be the state of $n_j$ for all $n_j \in \mathcal{N}(n_i)$;
    2. Let
           $x_i := f(x_j;\ n_j \in \{n_i\} \cup \mathcal{N}(n_i))$.
NOTIFY_i:
    1. Send $x_i$ to $p_0$;
    2. Send $x_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

In these procedures, x_i is a variable local to p_i.

5.3. FURTHER PROPERTIES IN THE ONE-DIMENSIONAL CASE

As we remarked in Section 5.1, in many cases the evolution of a cellular automaton according to its updating function leads to a limit cycle, or more particularly to a fixed point, in the state space of the automaton. This is always the case with (deterministic) finite cellular automata, but can also be observed in the behavior of (deterministic) infinite cellular automata. Although of difficult characterization in the general case, the presence, number, and length of limit cycles can for some simple cases be identified analytically. Far from dispensing with the need for computer simulation of these automata, the possibility of analytical treatment, even if limited, may turn out to be useful within the very scope of the simulation, as the analysis may yield clues to guide the termination detection of the simulation, which, as we know from Section 5.2, often depends on the detection of periodic behavior.

In this section, we discuss how to approach the analysis of limit cycles of cylindrical, one-dimensional cellular automata. The assumption of periodic boundary conditions is not essential, as the analysis extends easily to the case in which the one-dimensional cellular automaton is not cylindrical; it does, nevertheless, allow us to simplify the exposition. Our goal is to compute the number of states in limit cycles of length p, for some $p \geq 1$, of a cylindrical, one-dimensional cellular automaton. As customary in the one-dimensional case, the state of this cellular automaton may be viewed as a *string* with n elements, each representing the state of a node in the automaton, therefore a member of D. Adjacent elements in the string correspond to adjacent nodes in the cellular automaton's underlying lattice.

Our first step is to recognize that a string is in a limit cycle of length p for a cellular automaton under updating function f if and only if it is a fixed point for another cellular automaton with the same underlying lattice and updating function f_p, which is the pth composition of f, that is,

$$f_p \equiv f \circ \cdots \circ f,$$

where $\circ$ appears $p-1$ times. As a consequence, the number of strings in limit cycles of length p under f is the number of fixed points under f_p. The radius of the updating function f_p is pr, and a string is a fixed point under f_p if and only if, for all nodes in N,

$$f_p(y_{-pr} \cdots y_0 \cdots y_{pr}) = y_0, \tag{5.6}$$

where $y_{-pr} \cdots y_0 \cdots y_{pr}$ is the portion of the string centered at the node whose state is y_0. We assume that $2pr + 1 \leq n$.

The condition expressed in (5.6) is an invariance condition, and can be used to compose a $d^{2pr} \times d^{2pr}$ matrix M_p as follows. Suppose a row of M_p is indexed by an integer whose representation in the base d is $z_{-pr} z_{-pr+1} \cdots z_{pr-1}$. Likewise, let a column of M_p be indexed by the base-d integer $z'_{-pr+1} \cdots z'_{pr-1} z_{pr}$. Then, letting m denote the element of M_p corresponding to this row and this column, we define

$$m = \begin{cases} 1, & \text{if } z_{-pr+1} = z'_{-pr+1}, \ldots, z_{pr-1} = z'_{pr-1}, \\ & \quad \text{and } z_{-pr} \cdots z_0 \cdots z_{pr} \text{ satisfies (5.6);} \\ 0, & \text{otherwise.} \end{cases} \tag{5.7}$$

Theorem 5.1, given next, relates the number of fixed points under f_p to the sum of the diagonal elements of M_p^n, the *trace* of M_p^n.

Theorem 5.1. *The number of fixed points under the updating function f_p is given by the trace of M_p^n.*

Proof: Consider a directed graph whose nodes are base-d integers with $2pr$ digits. Suppose that a directed edge in this graph goes from a node to another if and only if the least significant $2pr-1$ digits of the former are pairwise equal to the most significant $2pr-1$ digits of the latter, and furthermore the string obtained by concatenating the least significant digit of the latter to the $2pr$ digits of the former satisfies (5.6) (Figure 5.4). Now create a matrix with one row and one column per node of this graph, and let each of its elements be assigned value 1 if an edge exists from the node corresponding to its row to the node corresponding to its column; the element is assigned value 0 otherwise. Clearly, this matrix is M_p, so the number of directed paths with $k \geq 1$ edges in the graph connecting two nodes is given by the corresponding element in M_p^k. In particular, if the two nodes are the same node,

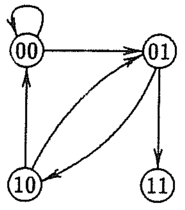


Figure 5.4. *For $r = 1$ and $p = 1$, this figure shows a directed graph having a node for each of the four base-2 integers with two digits* 00, 01, 10, *and* 11. *A directed edge exists from a node to another only if the least significant bit of the former is equal to the most significant bit of the latter (the converse need not be true).*

then this element (which is then a diagonal element) gives the number of directed cycles with k edges going through that node.

Now, a directed cycle with n edges in this graph can be interpreted as follows. Choose a node on the cycle and, starting at this node, form a string with the $2pr$th digits of each node (counted in increasing order of significance so that the least significant digit is the first). This string is clearly a fixed point under the updating function f_p, and is the same regardless of the first node we pick when building it, due to the assumed periodic boundary conditions. As a consequence, each directed cycle with n edges can be identified with a fixed point under f_p, and then the trace of M_p^n is the number of such fixed points. ■

The trace of a square matrix, defined as the sum of its diagonal elements, is also given by the sum of the *eigenvalues* of the matrix, which in turn are the roots of the *characteristic polynomial* of the matrix. The characteristic polynomial of M_p^n is obtained from the determinant of $\lambda I - M_p^n$, where I is the identity matrix (i.e., it has 1's in the main diagonal and 0's everywhere else), so the eigenvalues of M_p^n are the d^{2pr} solutions to the equation

$$\det(\lambda I - M_p^n) = 0. \tag{5.8}$$

Now, a well-known property of the eigenvalues of a matrix implies that, if λ_1 is an eigenvalue of M_p, then λ_1^n is an eigenvalue of M_p^n. As a consequence, the trace of M_p^n is given by

$$\sum_{k=1}^{d^{2pr}} \lambda_k^n, \tag{5.9}$$

where $\lambda_1, \ldots, \lambda_{d^{2pr}}$ are the eigenvalues of M_p, so it suffices to solve (5.8) for $n = 1$, i.e.,

$$\det(\lambda I - M_p) = 0. \tag{5.10}$$

Let us consider a simple example with $d = 2$, $r = 1$, and $p = 1$. Suppose, utilizing the notation style of (5.6), that f is such that

$$f(y_{-1}y_0y_1) = 0$$

for $y_{-1}y_0y_1 \in \{000, 001, 100, 101, 110, 111\}$, and such that

$$f(y_{-1}y_0y_1) = 1$$

for $y_{-1}y_0y_1 \in \{010, 011\}$. In this case, (5.6) is satisfied if and only if $y_{-1}y_0y_1 \in \{000, 001, 010, 011, 100, 101\}$, and M_1 is, by (5.7), the 4×4 matrix

$$M_1 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

By (5.10), the nonzero eigenvalues of M_1 are solutions to

$$\lambda^2 = \lambda + 1,$$

or, equivalently,

$$\lambda^n = \lambda^{n-1} + \lambda^{n-2},$$

so (5.9) yields

$$\sum_{k=1}^{4} \lambda_k^n = \sum_{k=1}^{4} \lambda_k^{n-1} + \sum_{k=1}^{4} \lambda_k^{n-2} \tag{5.11}$$

as the trace of M_1^n, where $\lambda_1, \ldots, \lambda_4$ are the eigenvalues of M_1. Clearly, the two summations in the right-hand side of (5.11) represent the traces of M_1^{n-1} and M_1^{n-2}, respectively, so for increasing values of n the traces of M_1^n constitute the so-called *generalized Fibonacci series* having as bases the traces of M_1 and M_1^2, respectively equal to 1 and 3.

The technique we illustrated by means of this example can then be used to yield more information than we sought initially (the number of strings in limit cycles of length p). By obtaining recurrence relations like the one in (5.11), we can also have the values of n for which limit cycles of length p exist, and the number of such limit cycles as well.

5.4. INTRODUCING PROBABILISTIC UPDATING FUNCTIONS

Probabilistic updating functions provide the possibility of chaotic (or at least nonperiodic) behavior for finite cellular automata. Such functions will be commonplace in two types of PC automaton networks, namely Markov Random Fields and Bayesian Networks, to be discussed respectively in Chapters 9 and 10. In the context of FC automaton networks, on the other hand, probabilistic behavior will be limited to this section and discussed rather briefly.

Let us first introduce a little notation. A distinction will be made between a node n_i's state at time $s \geq 0$, $x_i(s)$, now regarded as a random variable, and its value $d_i \in D$ (the value of the variable). A point in D^n, i.e., one of the possible states of the finite cellular automaton, will be denoted by explicitly listing its components, as in $(d_1, \dots, d_n)$. The notation $x_j(s-1) = d_j;\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)$ stands for the joint occurrence in a point in D^n at time $s - 1 \geq 0$ of d_j for all nodes n_j such that $n_j \in \{n_i\} \cup \mathcal{N}(n_i)$.

In finite cellular automata with probabilistic updating functions, (5.1) is replaced with an assignment to $x_i(s)$, for all $n_i \in N$ and all $s > 0$, of a value $d_i \in D$ chosen according to the conditional probability

$$P_i(s, d_i, d'_1, \dots, d'_n) = \Pr\big(x_i(s) = d_i \mid x_j(s-1) = d'_j;\ n_j \in \{n_i\} \cup \mathcal{N}(n_i)\big). \quad (5.12)$$

Clearly, the probabilistic updating based on the conditional probabilities of (5.12) entails a discrete-time Markov chain with transition probabilities

$$\Pr\big(x_1(s) = d_1, \dots, x_n(s) = d_n \mid x_1(s-1) = d'_1, \dots, x_n(s-1) = d'_n\big)$$
$$= \prod_{i=1}^{n} P_i(s, d_i, d'_1, \dots, d'_n), \quad (5.13)$$

as the update decisions are at all nodes independent of one another.

If the conditional probabilities in (5.12) are the same for all $s > 0$ (and then consequently so are the transition probabilities in (5.13)), then the Markov chain is homogeneous. If in addition all the conditional probabilities in (5.12) are strictly positive, then so are the transition probabilities in (5.13), and the Markov chain admits a stationary distribution P which is strictly positive over all of D^n. This distribution is obtained by solving

$$P(d_1, \dots, d_n) = \sum_{(d'_1, \dots, d'_n) \in D^n} P(d'_1, \dots, d'_n) \prod_{i=1}^{n} P_i(d_i, d'_1, \dots, d'_n) \quad (5.14)$$

for all $(d_1, \dots, d_n) \in D^n$, subject to

$$\sum_{(d_1, \dots, d_n) \in D^n} P(d_1, \dots, d_n) = 1.$$

Note in (5.14) that we have omitted the dependency on s of the conditional probabilities P_i (cf. (5.12)) to stress the chain's homogeneity. Examples can be found in the literature for which P is determined based on a Boltzmann-Gibbs distribution on an $(n+1)$-dimensional discrete space. This distribution is also considered in Chapters 9 and 10, but in an entirely different context.

As a final remark, we mention that there are possibilities for the conditional probabilities in (5.12) that guarantee

$$P(d'_1, \dots, d'_n) \prod_{i=1}^{n} P_i(s, d_i, d'_1, \dots, d'_n) = P(d_1, \dots, d_n) \prod_{i=1}^{n} P_i(s, d'_i, d_1, \dots, d_n)$$

for all $(d_1, \dots, d_n), (d'_1, \dots, d'_n) \in D^n$. As we know, this characterizes the reversibility of the Markov chain, and means intuitively that the automaton's evolution in time under the stationary distribution P is the same in both directions of the progress of time.

5.5. BIBLIOGRAPHIC NOTES

Cellular automata date back to their introduction by von Neumann and Ulam (von Neumann, 1966). Various of their aspects are discussed in Wolfram (1986). The four-category classification of cellular automata discussed in Section 5.1 is from Wolfram (1984), and is also considered by Jen (1986a). Additional material on cellular automata is given by Goles and Martínez (1990).

The material in Section 5.3 is based on Jen (1989), and is derived from an earlier treatment in which periodic boundary conditions were not considered (Jen, 1986b).

Section 5.4 is entirely based on Lebowitz, Maes, and Speer (1990).

6

Analog Hopfield neural networks

Analog Hopfield neural networks are the subject of this chapter. They are introduced, along with relevant properties, in Section 6.1. Simulation algorithms are given in Section 6.2, after a characterization of the networks as FC automaton networks. The Hopfield-Tank model for combinatorial optimization is introduced in Section 6.3, where it is illustrated by the use of analog Hopfield neural networks to solve the traveling salesman problem. Bibliographic notes are given in Section 6.4.

6.1. THE MODEL AND BASIC PROPERTIES

In an analog Hopfield neural network, each member n_i of N is viewed as an *artificial neuron* (or simply *neuron*). At time $t \geq 0$, the *state* $v_i(t)$ of n_i is a real number in the interval $[0, 1]$, and is given by the *sigmoid* ("S"-shaped) transfer function

$$\begin{aligned} v_i(t) &= g_i\big(u_i(t)\big) \\ &= \frac{1}{1 + \exp\Big(-\gamma_i\big(u_i(t) - \theta_i\big)\Big)}. \end{aligned} \tag{6.1}$$

In (6.1), $\gamma_i \geq 0$ is the neuron's *gain*, $u_i(t)$ its *potential*, and θ_i its *threshold potential* (or simply *threshold*) (Figure 6.1).

For $t \geq 0$, the potential of neuron n_i evolves according to the linear, first-order, ordinary differential equation

$$C_i \frac{du_i(t)}{dt} = \sum_{j=1}^{n} w_{ij} v_j(t) + e_i - \frac{u_i(t)}{R_i}, \tag{6.2}$$

where

$$\sum_{j=1}^{n} w_{ij} v_j(t) + e_i$$

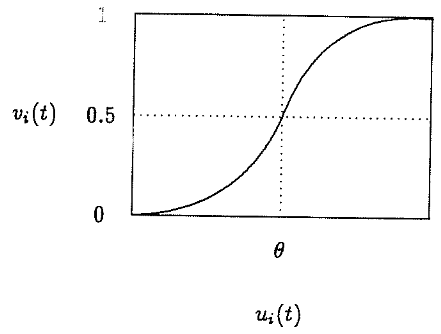


Figure 6.1. *The sigmoid transfer function is such that $v_i(t)$ approaches 0 for $u_i(t) \ll \theta_i$ and 1 for $u_i(t) \gg \theta_i$. For $u_i(t) = \theta_i$, it yields $v_i(t) = 0.5$. The function shown here has $\gamma_i = 1$.*

is the *input* to n_i.

In (6.2), C_i is the *capacitance* of n_i, R_i its *resistance*, e_i an input to the neuron coming from outside the network (an *external input*), and w_{ij} is the strength of the influence of neuron n_j upon neuron n_i. By analogy with the terminology used when dealing with natural neurons, w_{ij} is called the *synaptic strength* from n_j to n_i. The system of coupled differential equations given for $n_i \in N$ by (6.2) models the neural network as an electric circuit in which the u_i's and v_i's are voltages, the e_i's are currents, and the w_{ij}'s are conductances.

As the neurons' states evolve in time as given by the differential equations of (6.2), the *state* of the network, given by the assembled states of the n neurons, evolves as well. However, the global behavior of the network is difficult to characterize, as the differential equations are very intimately coupled with one another and do not directly yield any clue to the neurons' collective properties. Consider, nevertheless, the function

$$E(t) = -\sum_{i=1}^{n}\sum_{j=i}^{n} w_{ij}v_i(t)v_j(t) - \sum_{i=1}^{n} e_i v_i(t) + \sum_{i=1}^{n} \frac{1}{R_i}\int_{y=0}^{v_i(t)} g_i^{-1}(y)dy, \qquad (6.3)$$

called the *energy* of the network. This function, as we demonstrate in Theorem 6.1 given next, is the key to very important global properties of the analog Hopfield neural network, and accounts for most of its applications, as we discuss later in this chapter.

Theorem 6.1. *If $w_{ij} = w_{ji}$ for all $n_i, n_j \in N$, then the set of differential equations in (6.2) for all $n_i \in N$ implies*

$$\frac{dE(t)}{dt} \leq 0$$

for all $t \geq 0$.

Proof: Because of the symmetry $w_{ij} = w_{ji}$ for all $n_i, n_j \in N$, the time-derivative of (6.3) is given by

$$\frac{dE(t)}{dt} = -\sum_{i=1}^{n} \frac{dv_i(t)}{dt}\left(\sum_{j=1}^{n} w_{ij}v_j(t) + e_i - \frac{u_i(t)}{R_i}\right),$$

where the term in parentheses is easily recognized as the right-hand side of (6.2), and therefore

$$\begin{aligned}\frac{dE(t)}{dt} &= -\sum_{i=1}^{n} C_i \frac{dv_i(t)}{dt}\frac{du_i(t)}{dt} \\ &= -\sum_{i=1}^{n} C_i \frac{dv_i(t)}{du_i(t)}\left(\frac{du_i(t)}{dt}\right)^2,\end{aligned}$$

which is nonpositive by (6.1). ■

Theorem 6.1 means that the evolution in time of the neural network according to (6.2) leads to local minima of the energy $E(t)$ given by (6.3). The energy then acts as a Lyapunov function of the dynamic system represented by the neural network, as the system's stable states (or fixed points) are minima of the energy. Such minima are then referred to as *stable states* of the neural network (Figure 6.2). Applications of the analog Hopfield neural network rely heavily on this fact, as for example associative memories and the solution of optimization problems. In the former case, items to be stored in the memory are identified with minima of $E(t)$. In the latter case, $E(t)$ is associated with the objective function of the optimization problem, whose minima are then minima of $E(t)$. We relegate the application to associative memories to Chapter 8. The application to optimization is treated in Section 6.3 in the context of the traveling salesman problem, a notoriously difficult optimization problem, and in Chapters 8 and 9 as well.

Neuron n_i's gain, the γ_i of (6.1), regulates the steepness of the sigmoid function around the potential $u_i = \theta_i$. If γ_i is very small, the curve is nearly flat, whereas the larger γ_i is the closer the curve resembles a step at $u_i = \theta_i$. In the limit as $\gamma_i \to \infty$ for all $n_i \in N$, interesting properties are revealed. First of all, the case of high gains entails an important simplification in the form of the energy function of (6.3). Specifically, it is rather simple to see that

$$\lim_{\substack{\gamma_1 \to \infty \\ \cdots \\ \gamma_n \to \infty}} \int_{y=0}^{v_i(t)} g_i^{-1}(y)dy = \theta_i v_i(t)$$

for all $t \geq 0$ and all $n_i \in N$, so

$$\lim_{\substack{\gamma_1 \to \infty \\ \cdots \\ \gamma_n \to \infty}} E(t) = -\sum_{i=1}^{n}\sum_{j=i}^{n} w_{ij}v_i(t)v_j(t) - \sum_{i=1}^{n} e_i v_i(t) + \sum_{i=1}^{n} \frac{\theta_i}{R_i} v_i(t) \qquad (6.4)$$

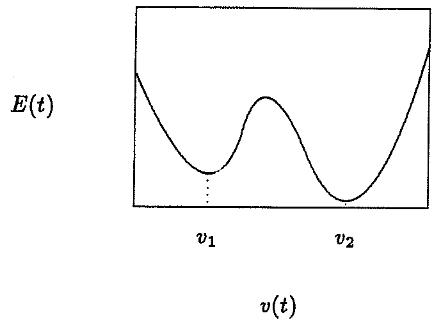


Figure 6.2. *Local minima of $E(t)$, as v_1 and v_2 in this figure, are stable states of the neural network.*

for all $t \geq 0$. This will be of great relevance when we discuss applications to optimization in Section 6.3.

In addition to this property, under high gains the network's stable states comprise solely 0's and 1's as individual neuron states, as established by Theorem 6.2.

Theorem 6.2. *If $w_{ij} = w_{ji}$ for all $n_i, n_j \in N$, then, in the limit as $\gamma_i \to \infty$ for all $n_i \in N$, every stable state is in $\{0,1\}^n$.*

Proof: This theorem follows directly from the fact that (6.4) is identical to (8.2), which gives the energy of a binary Hopfield neural network (to be discussed later), whose stable states are minima of (8.2) and all in $\{0,1\}^n$. ∎

Theorem 6.2 too has a great appeal in the context of applying analog Hopfield neural networks to the solution of optimization problems, as the neurons' states can be regarded as variables whose values can only be one of two possibilities at the feasible points. We shall return to this point shortly in Section 6.3.

6.2. SIMULATION ALGORITHMS

6.2.1. The sequential algorithm

The differential equations given by (6.2) for all $n_i \in N$, together with the initial values of the neurons' potentials ($u_i(0)$ for all $n_i \in N$), characterize what is known as an *initial-value problem.* Initial value problems are in general of difficult analytical solution, especially when consisting of a system of coupled differential equations, as is the case of (6.2). The usual approach is then to solve the equation or system of equations numerically.

Let us consider the generic initial-value problem on the functions $y_1(t), \ldots, y_n(t)$ given by

$$\frac{dy_i(t)}{dt} = \phi_i(y_1(t), \ldots, y_n(t)) \tag{6.5}$$

for $1 \leq i \leq n$, along with the initial values $y_1(0), \ldots, y_n(0)$. The simplest numerical method to solve the system of differential equations given in (6.5) is *Euler's method*, which provides an approximation of $y_1(t), \ldots, y_n(t)$ at the discrete values $\Delta t, 2\Delta t, \ldots$ of t according to

$$y_i(\tau) = y_i(\tau - \Delta t) + \Delta t \phi_i(y_1(\tau - \Delta t), \ldots, y_n(\tau - \Delta t)), \tag{6.6}$$

for all $n_i \in N$ and $\tau \in \{\Delta t, 2\Delta t, \ldots\}$. (6.6) is in fact a linear approximation of the first-order derivative with respect to t of $y_i(t)$ at $t = \tau - \Delta t$.

The differential equation in (6.2) is of course of the form of the one in (6.5), so (6.6) can be applied directly to solve numerically the system of differential equations given for all $n_i \in N$ by (6.2). In this case, Euler's method allows the potential $u_i(t)$ to be approximated for the discrete times $\tau \in \{\Delta t, 2\Delta t, \ldots\}$ given the initial value $u_i(0)$ for all $n_i \in N$. This is achieved by applying (6.6), which then becomes

$$u_i(\tau) = u_i(\tau - \Delta t) + \frac{\Delta t}{C_i}\left(\sum_{j=1}^{n} w_{ij} v_j(\tau - \Delta t) + e_i - \frac{u_i(\tau - \Delta t)}{R_i}\right), \tag{6.7}$$

for all $n_i \in N$.

The application of Euler's method to solve the initial-value problem specified by the system of differential equations given by (6.2) for all $n_i \in N$ allows us to view an analog Hopfield neural network as an FC automaton network. By letting $s = \tau/\Delta t$, we obtain an FC automaton network where G's edge set has (n_i, n_j) as a member if and only if $w_{ij} \neq 0$, and where the updating function f, based on (6.7), is such that $x_i(0) = v_i(0)$ and $x_i(s) = v_i(\tau)$ for all $n_i \in N$ and all $s > 0$ (cf. Chapter 1). A sequential algorithm to simulate this FC automaton network for discretized values of t is then as given next, following (6.7). The algorithm runs until no neuron has its state updated to a value that is farther from the current value by more than ε, for some $\varepsilon > 0$.

Algorithm *Seq_Analog_Hopfield:*

```
$\tau := 0$;
repeat
    $\tau := \tau + \Delta t$;
    for $i := 1$ to $n$ do
        $u_i(\tau) := u_i(\tau - \Delta t)$
            $+\frac{\Delta t}{C_i}\left(\sum_{j=1}^{n} w_{ij}v_j(\tau - \Delta t) + e_i - \frac{u_i(\tau - \Delta t)}{R_i}\right)$
until $|v_i(\tau) - v_i(\tau - \Delta t)| \le \varepsilon$ for all $n_i \in N$.
```

Another method to solve numerically the system of differential equations of (6.5) is the *Runge-Kutta method.* This method too works on discrete time intervals Δt to generate approximations at the instants $\tau \in \{\Delta t, 2\Delta t, \ldots\}$, but it generally outperforms Euler's method in terms of the accuracy of the approximations obtained. In contrast with Euler's method, the Runge-Kutta method employs a linear combination of approximations of the first-order derivative with respect to t of $y_i(t)$ at $t = \tau - \Delta t$ for $\tau \in \{\Delta t, 2\Delta t, \ldots\}$. This is achieved with

$$y_i(\tau) = y_i(\tau - \Delta t) + \frac{\Delta t}{6}(K_1^i + 2K_2^i + 2K_3^i + K_4^i), \tag{6.8}$$

where

$$\begin{aligned}
K_1^i &= \phi_i(y_1(\tau - \Delta t), \ldots, y_n(\tau - \Delta t))\\
K_2^i &= \phi_i\left(y_1(\tau - \Delta t) + \frac{K_1^1}{2}, \ldots, y_n(\tau - \Delta t) + \frac{K_1^n}{2}\right)\\
K_3^i &= \phi_i\left(y_1(\tau - \Delta t) + \frac{K_2^1}{2}, \ldots, y_n(\tau - \Delta t) + \frac{K_2^n}{2}\right)\\
K_4^i &= \phi_i\left(y_1(\tau - \Delta t) + K_3^1, \ldots, y_n(\tau - \Delta t) + K_3^n\right),
\end{aligned} \tag{6.9}$$

for all $n_i \in N$.

When applied to the system of (6.2), (6.8) and (6.9) yield, respectively,

$$u_i(\tau) = u_i(\tau - \Delta t) + \frac{\Delta t}{6}(K_1^i + 2K_2^i + 2K_3^i + K_4^i) \tag{6.10}$$

and

$$\begin{aligned}
K_1^i &= \frac{1}{C_i}\left(\sum_{j=1}^{n} w_{ij}v_j(\tau - \Delta t) + e_i - \frac{u_i(\tau - \Delta t)}{R_i}\right)\\
K_2^i &= \frac{1}{C_i}\left(\sum_{j=1}^{n} w_{ij}g_j\left(u_j(\tau - \Delta t) + \frac{K_1^j}{2}\right) + e_i - \frac{1}{R_i}\left(u_i(\tau - \Delta t) + \frac{K_1^i}{2}\right)\right)\\
K_3^i &= \frac{1}{C_i}\left(\sum_{j=1}^{n} w_{ij}g_j\left(u_j(\tau - \Delta t) + \frac{K_2^j}{2}\right) + e_i - \frac{1}{R_i}\left(u_i(\tau - \Delta t) + \frac{K_2^i}{2}\right)\right)\\
K_4^i &= \frac{1}{C_i}\left(\sum_{j=1}^{n} w_{ij}g_j\left(u_j(\tau - \Delta t) + K_3^j\right) + e_i - \frac{1}{R_i}\left(u_i(\tau - \Delta t) + K_3^i\right)\right).
\end{aligned} \tag{6.11}$$

Analogously to what we did with Euler's method, a sequential algorithm can also be easily obtained to solve the system of differential equations of (6.2) by means of the Runge-Kutta method. For such, it suffices to utilize (6.11) and (6.10), in this order, in the innermost iterative loop of Algorithm *Seq_Analog_Hopfield.* If (6.11) is applied in the order shown, then at each step the components needed to proceed will have already been calculated. The Runge-Kutta method then provides us with another view of an analog Hopfield neural network as an FC automaton network. This view is entirely analogous to the one we discussed based on Euler's method, but employs a different updating function f, now based on (6.10) and (6.11).

6.2.2. The distributed parallel algorithm

We saw in Section 6.2.1 that an analog Hopfield neural network can be regarded as an FC automaton network when the solution of its equations is approached via Euler's method or the Runge-Kutta method. This is so because these two methods (as well as others) work by approximating the neurons' states at discretized times, and at each such time only the states obtained in the previous iteration are employed.

Once an analog Hopfield neural network is viewed as an FC automaton network, its distributed parallel simulation follows the template given in Section 4.3.2. In this case, processor p_0 participates as a detector of the convergence of the simulation, as determined by the parameter ε discussed in Section 6.2.1. In order to fill the template, we must specify the procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for every processor p_i such that $n_i \in N$. Recall from Section 4.3.2 that MSG_i contains exactly one message from each of p_i's neighbors. Each one of these messages is the state of the corresponding neuron to be used in updating n_i's state.

The procedures we provide are based, as Algorithm *Seq_Analog_Hopfield,* on (6.7). But it should be clear from our discussion in Section 6.2.1 that these procedures might as well employ (6.10) and (6.11) without any difficulty. The main steps of each procedure are given next.

```
INITIALIZE_i:
  1. Let $u_i := u_i(0)$;
  2. Let $v_i := g_i(u_i)$;
  3. Send $v_i$ to $p_0$;
  4. Send $v_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
UPDATE_i(MSG_i):
  1. Let $v_j \in MSG_i$ be the state of $n_j$ for all $n_j \in \mathcal{N}(n_i)$;
  2. Let
     $u_i := u_i + \frac{\Delta t}{C_i}\left(\sum_{j=1}^{n} w_{ij}v_j + e_i - \frac{u_i}{R_i}\right)$;
  3. Let $v_i := g_i(u_i)$.
NOTIFY_i:
  1. Send $v_i$ to $p_0$;
  2. Send $v_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

In these procedures, v_i and u_i are variables local to p_i.

6.3. THE TRAVELING SALESMAN PROBLEM

The Traveling Salesman Problem (TSP) is stated as follows. Given a set of cities C, let a tour of these cities be called *feasible* if and only if it includes every city exactly once. If every city can be reached directly from every other city via a connection of known length, then TSP asks for a feasible tour of minimum length. This problem can be thought of as a problem on a completely connected undirected graph, in which a node is a city and an undirected edge is a direct connection between two cities. Edges are then weighted to represent the lengths of the various connections. Viewed like this, TSP asks for a Hamiltonian cycle of minimum length on the graph.

For any two cities $a, b \in C$, the distance between a and b is $d_{ab} = d_{ba}$. We say that the distances between cities satisfy the *triangle inequality* if and only if

$$d_{ab} \leq d_{ac} + d_{cb}$$

for all $a, b, c \in C$.

TSP is an $\mathcal{NP}$-hard problem (even if distances are required to satisfy the triangle inequality), which as we know indicates that an efficient algorithm to solve it exactly is very unlikely to be found. In addition to this inherent intractability, TSP has over the years acquired a paradigmatic status among the difficult problems of combinatorial optimization, as several real-world problems have similar formulations. TSP is then a natural candidate to test heuristics for the approximation of optimal solutions to hard combinatorial problems. In the case of TSP, two heuristics have become especially famous, one for its success in practical situations (the Lin-Kernighan heuristic), the other for its elegant theoretical properties (the Christofides heuristic). Both heuristics work on the view of TSP as a problem on a completely connected undirected graph.

The Lin-Kernighan heuristic is based on the notion of k-optimality for $0 \leq k \leq |C|$, which in the context of TSP means the following. Let T be a feasible tour, and T' an infeasible tour obtained from T by removing k edges (T' has then $|C| - k$ edges). Make all possible extensions of T' that yield a feasible tour, and let the shortest tour found replace T, if shorter than T. Then repeat the process until no more combinations of k edges remain in T that were not removed. The shortest tour found along the entire process is k-optimal. The Lin-Kernighan heuristic searches for k-optimal tours starting at an initial tour, without however fixing the value of k *a priori*. Instead, at each repetition of the process, the value of k is picked as the largest integer in the appropriate range that can yield an improvement in the tour length.

The Christofides heuristic starts by determining a minimum spanning tree with the inter-city distances. This tree has, of course, an even number of odd-degree nodes, so the completely connected subgraph on these nodes has at least one complete matching. The next step is to find the minimum complete matching, and then to use its edges, along with those of the minimum spanning tree, to create a multigraph on the $|C|$ nodes. This multigraph is guaranteed to be Eulerian, as all of its nodes have even degrees (odd-degree nodes in the minimum spanning tree have in the multigraph an additional adjacent edge given by the minimum complete matching). An Eulerian cycle can then be found in it, and embedded in this cycle a tour of the $|C|$ cities on the original graph. What is remarkable about this heuristic is that the resulting tour is never longer than the shortest tour by more than fifty percent if the distances satisfy the triangle inequality. We shall encounter a similar type of behavior in Chapter 8 when we discuss the minimum node cover problem.

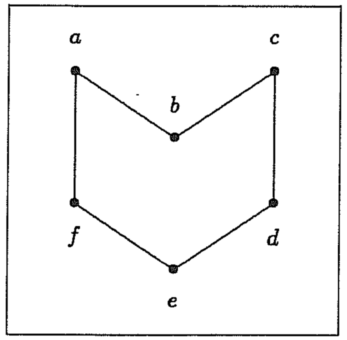
TSP has also been used to demonstrate the efficacy (or lack thereof, in some cases) of what has become known as the *Hopfield-Tank model* for combinatorial optimization. We describe this model within the context of solving TSP, and use the analog Hopfield neural network as the basic mechanism of solution. It is possible, however, to apply the model to many other problems in combinatorial optimization (another example is discussed in Section A.3), and to utilize other mechanisms of solution, as the ones described in Chapters 8 and 9.

The essence of the Hopfield-Tank model is to regard the neurons' states as variables of a combinatorial optimization problem, and to assign the neural network's parameters values that make its energy equivalent to the objective function of the problem under consideration. As by Theorem 6.1 the network's energy $E(t)$ is continuously nonincreasing as the network evolves, then so is the objective function, and hopefully when the network stabilizes a good solution to the combinatorial optimization problem will have been obtained.

The basic idea in the case of TSP is to employ $n = |C|^2$ neurons arranged as a $|C| \times |C|$ two-dimensional array. Each row in this array corresponds to one of the $|C|$ cities, while each column corresponds to one of the $|C|$ positions a city may ocupy in a feasible tour (Figure 6.3). Differently from what we have been doing from the beginning of this chapter, we shall for convenience denote by n_{cr} the neuron occupying in the array the row corresponding to city $c \in C$ and the column corresponding to position r, $1 \leq r \leq |C|$. A feasible tour at some time $t \geq 0$ is represented in this array of neurons by a situation of neuron states such that

$$v_{cr}(t) \approx \begin{cases} 1, & \text{if } c \text{ is the } r\text{th city in the tour;} \\ 0, & \text{otherwise} \end{cases}$$

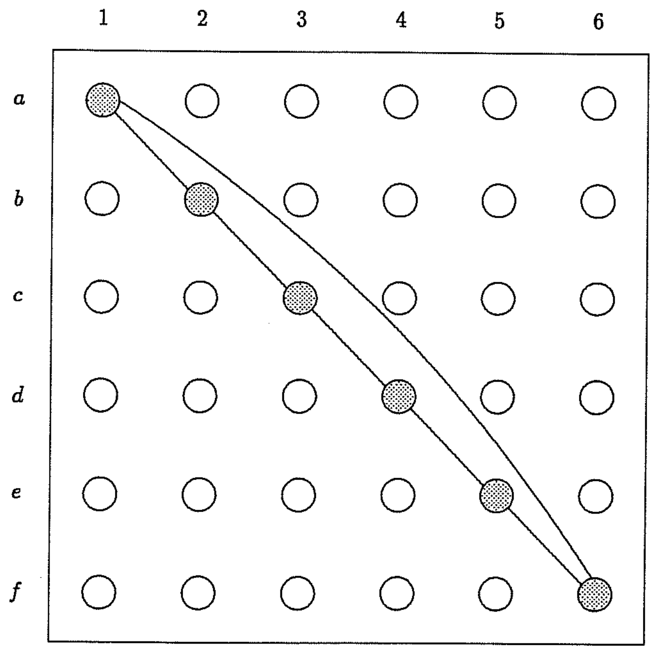
for all $c \in C$ and all r such that $1 \leq r \leq |C|$. As a consequence, it must be that cities in C and integers in the range $[1, |C|]$ be paired in such a way that every city and every integer participates in exactly one pair. In the array of neurons, this corresponds to a situation in which exactly $|C|$ neurons have states close to 1, of which exactly one per row and one per column. The states of all other neurons are close to 0.



(a)

Figure 6.3. *In part (a), we show a TSP tour for a six-city instance of the problem. In part (b), the $|C| \times |C|$ (in this case, 6×6) array of neurons to solve TSP is shown in a stable state corresponding to the tour of part (a). Shaded neurons are in states close to 1, and the others are in states close to 0. Only the connections between neurons with states close to 1 are shown.*

These constraints can be thought of as having a "syntactic" nature, as they deal with the feasibility of TSP tours in the proposed model. We proceed as follows in order to ensure that they are satisfied. First of all, we employ neurons with high-gain sigmoid transfer functions, i.e., high values of γ_{cr} for all $c \in C$ and all r such that $1 \leq r \leq |C|$. By Theorem 6.2, this is a guarantee that neurons will all be at states either very near 0 or very near 1 when the network stabilizes. Secondly, every two neurons on the same row or on the same column in the array are interconnected



(b)

Figure 6.3 (continued)

by a synaptic strength of negative sign and very high magnitude, $W < 0$. These are meant to inhibit stable states of the network in which more than one neuron's state is close to 1 on a same row or column. Intuitively, $|W|$ has to be as large as needed for the (positive) contribution to $E(t)$ of the pair of neurons with states approaching 1 to be so large that very hardly will such a network state be stable. However, we must also avoid situations in which no neuron in a row or column stabilizes with its state close to 1. Assigning each neuron's threshold a negative value is then how to enforce that not all neurons are "turned off." If this value is "negative enough," then each neuron contributes to $E(t)$ with a negative amount of large magnitude, and as a consequence it is unlikely that a network state with entire rows or columns of 0's is stable. These contributions of both W and the neurons' thresholds to $E(t)$ can be easily understood by a quick examination of (6.4), which in high-gain situations gives the network's energy.

Having dealt with the feasibility of stable states, the remaining parameters of the neural network are set as follows. For $a, b \in C$ and $1 \leq r, s \leq |C|$, with $a \neq b$ and $r \neq s$, the synaptic strength between neurons n_{ar} and n_{bs} is nonzero and given by

$$w_{ar,bs} = -d_{ab}$$

if and only if either $s = r+1$ or $s = 1$ and $r = |C|$ (i.e., columns r and s are adjacent to each other, given that columns 1 and $|C|$ are considered adjacent to each other; cf. Figure 6.3).

For the sake of simplicity, let us assume that all neurons have the same threshold and the same resistance, respectively θ and R. Theorem 6.3, given next, provides a sufficient condition on the neural network parameters we have just described for the network to be in a feasible state whenever it stabilizes. By a *feasible state* of the neural network we mean a state that represents a feasible TSP tour.

Theorem 6.3. *If*

$$W < \frac{\theta}{R} < \min_{a \in C}\left\{-2 \sum_{b \in C | b \neq a} d_{ab}\right\},$$

then every stable state of the neural network is feasible.

Proof: Suppose, to the contrary, that for some time $t \geq 0$ a stable state of the neural network is infeasible. Then one of the following three cases must happen.

(i) There are $|C|$ neurons with states close to 1, but not one per row and one per column.

(ii) There are more than $|C|$ neurons with states close to 1.

(iii) There are less than $|C|$ neurons with states close to 1.

If case (i) or case (ii) holds, then at least one row (say c) has $x \geq 1$ neurons with states close to 1, and at least one column (say r) has $y \geq 1$ neurons with states close to 1, such that $x + y > 2$ and $v_{cr}(t) \approx 1$. In this case, $u_{cr}(t) > \theta$, and then (6.2) yields

$$C_{cr}\frac{du_{cr}(t)}{dt} < (x + y - 2)W - \frac{\theta}{R},$$

where we have taken into account the strict negativity of the synaptic strengths that connect n_{cr} outside its row or column. By the first inequality in the hypothesis, and considering that $x + y > 2$, it then follows that

$$\frac{du_{cr}(t)}{dt} < 0.$$

Similarly, if case (iii) holds, then there must exist one row (say c) and one column (say r) with no neuron at all whose state is close to 1. In this case, $u_{cr}(t) < \theta$, and then, by (6.2),

$$C_{cr}\frac{du_{cr}(t)}{dt} > -2 \sum_{b \in C | b \neq c} d_{bc} - \frac{\theta}{R}.$$

By the second inequality in the hypothesis, we then have

$$\frac{du_{cr}(t)}{dt} > 0.$$

In conclusion, we have a neuron n_{cr} for which either $u_{cr}(t) > \theta$ and $du_{cr}(t)/dt < 0$ or $u_{cr}(t) < \theta$ and $du_{cr}(t)/dt > 0$. The neural network state must then be unstable. ■

It is a consequence of Theorem 6.3, by (6.4), that the network's energy at stable states is given by the length of a feasible tour plus the constant $|C|\theta/R$. Another consequence is that the optimal tour corresponds to a stable state of the neural network, so TSP is equivalent to finding a global minimum of $E(t)$. Unfortunately, the evolution in time of an analog Hopfield neural network leads to local minima of $E(t)$, by Theorem 6.1, so not always is a good TSP tour obtained as a solution. We shall, in Chapter 9, see how local minima can be avoided when the neurons' states are binary, i.e., restricted to be either 0 or 1.

6.4. BIBLIOGRAPHIC NOTES

The analog Hopfield neural network was introduced by Hopfield (1984), on whose work the material in Section 6.1 is based, and falls within the more general class of analog neural networks treated by Mead (1989). Two additional papers that may also be referred to are de Carvalho and Barbosa (1989), where a technique is presented to occasionally avoid the gradient-descent behavior of analog Hopfield neural networks, and Barbosa and de Carvalho (1991), where a learning algorithm for this model is presented.

Both Euler's method and the Runge-Kutta method, employed in Section 6.2 to solve the differential equations associated with analog Hopfield neural networks, can be found in Wylie (1975) and Braun (1978). The algorithm for distributed parallel simulation appeared in Barbosa and Lima (1990).

There are various sources to complement the material on TSP presented in Section 6.3. General literature on the problem, including its intractability aspects and heuristic approaches, includes Garey and Johnson (1979) and Papadimitriou and Steiglitz (1982). The Lin-Kernighan heuristic appeared in Lin and Kernighan (1973), employing techniques developed previously for other problems (Kernighan and Lin, 1970). The Christofides heuristic was introduced in Christofides (1976).

The Hopfield-Tank model for combinatorial optimization was introduced by Hopfield and Tank (1985, 1986), and very criticized by Wilson and Pawley (1988). One of the problems with the original proposal to apply the model to the solution of TSP appears to have been the proposed objective function. The one we employ in Section 6.3 is from de Carvalho and Barbosa (1990). Related work includes Brandt, Wang, Laub, and Mitra (1988), Hegde, Sweet, and Levy (1988), van den Bout and Miller (1988), Akiyama, Yamashita, Kajiura, and Aiso (1989), Takefuji and Szu (1989), and van den Bout and Miller (1989). Various other heuristics for TSP, similar in flavor to the use of neural networks, have appeared (see Peterson

(1990) for a comparative analysis). Of these, one that seems to be very successful is the so-called elastic-net method, introduced by Durbin and Willshaw (1987). The method was later analyzed in detail (Durbin, Szeliski, and Yuille, 1989; Simmen, 1991) and extended (Burr, 1988). A variation of the method especially calibrated for efficiency in both sequential and distributed parallel implementations was given by Boeres, de Carvalho, and Barbosa (1992).

7 Other analog neural networks

Analog neural networks similar to the analog Hopfield neural networks treated in Chapter 6 are the subject of this chapter. Two types of such neural networks are discussed, although the second one can be regarded as a variation of the first one. The first type is introduced for the solution of systems of linear algebraic equations in Section 7.1. The corresponding simulation algorithms, after a characterization of the network as an FC automaton network, are given in Section 7.2. The second type of neural network is discussed in Section 7.3 in the context of solving linear programs. Bibliographic notes are presented in Section 7.4.

7.1. SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS

Let m_1 and m_2 be positive integers. A *system of linear algebraic equations* (or simply *linear system*) with m_1 equations and m_2 variables is the system

$$\begin{gathered}\sum_{\ell=1}^{m_2} a_{k\ell} y_\ell = b_k; \\ 1 \le k \le m_1,\end{gathered} \tag{7.1}$$

where each $a_{k\ell}$ is an element of a matrix $A \in \mathbf{R}^{m_1 \times m_2}$, each b_k is an element of a vector $b \in \mathbf{R}^{m_1}$, and each y_ℓ is one of the m_2 variables to be determined. The linear system in (7.1) is then also succinctly denoted by

$$Ay = b, \tag{7.2}$$

where $y \in \mathbf{R}^{m_2}$ is the vector of elements y_ℓ for $1 \le \ell \le m_2$. Determining y in (7.2) is one of the most central problems in many sciences and in engineering.

Depending on the matrix A and on the vector b, the system in (7.2) may have no solution, exactly one solution, or infinitely many solutions. What is needed to decide for one of these three cases is the concept of the *rank* of a matrix, defined to be the maximum number of linearly independent columns of the matrix (a group of columns is linearly independent if and only if the only linear combination of them yielding the vector 0 is the one that employs zero coefficients exclusively). Let $A|b$ denote the matrix in $\mathbf{R}^{m_1\times(m_2+1)}$ obtained by adding b to A as an additional column. If the rank of A and the rank of $A|b$ are not the same, then the linear system in (7.2) has no solution. If they are the same, then the linear system has solutions, and the solution is unique if and only if the ranks are equal to m_2. If in addition $m_1 = m_2$, then the unique solution is given by $A^{-1}b$, where A^{-1} is the *inverse* of A. Infinitely many solutions exist otherwise.

Several methods of solution exist for the linear system in (7.2). Some of them are direct methods, as Gaussian elimination, others are iterative, as the Gauss-Seidel method. These methods are based on sequential algorithms, but variants of them (especially of the iterative methods) have been proposed for parallel machines. Although the problem and the methods have been investigated for many years, larger instances of the problem seem to appear almost continually within various fields of scientific investigation, so the interest in new methods and approaches never wanes. Particularly interesting are approaches that employ massive parallelism in a scalable fashion, as these can keep up with the ever-increasing problem sizes.

Given the many possibilities regarding the solutions of the linear system in (7.2), one natural possibility is to pose it as the unconstrained optimization problem on $\mathbf{R}^{m_2}$ that asks for the global minimum of the function $\epsilon : \mathbf{R}^{m_2} \to \mathbf{R}$ given by

$$\epsilon(y) = \frac{1}{2}(Ay-b)^T(Ay-b) \tag{7.3}$$

for all $y \in \mathbf{R}^{m_2}$, where the superscript T denotes the *transpose* of a matrix. This function is proportional to the square of the Euclidean distance between Ay and b in $\mathbf{R}^{m_1}$, and the problem to find its minima is the *least-squares problem*. The Hessian matrix of the function in (7.3) (this is the matrix of second-order derivatives of ϵ) is A^TA, so we have, for all $y \in \mathbf{R}^{m_2}$,

$$y^TA^TAy \geq 0,$$

meaning that the Hessian matrix is *semi-positive definite*, and that, consequently, the function in (7.3) is convex. Every local minimum of ϵ is then a global minimum as well.

The convexity of ϵ in (7.3) implies that its minima occur at points $y \in \mathbf{R}^{m_2}$ at which $\nabla\epsilon(y) = 0$, where

$$\nabla\epsilon(y) = A^T(Ay-b) \tag{7.4}$$

is the gradient of ϵ at y. For linear systems with at least one solution, this happens at points y such that $\epsilon(y) = 0$, so $Ay = b$ (that is, y is a solution to the linear system in (7.2)). If the linear system does not have any solution, on the other hand, then the zero-gradient condition happens at one or an infinity of points y such that $\epsilon(y) > 0$. Among these points, the (unique) point y^+ of minimum Euclidean norm

(i.e. $(y^+)^Ty^+ \leq y^Ty$ for all $y \in \mathbf{R}^{m_2}$ such that $\nabla\epsilon(y) = 0$) is given by the *pseudo-inverse* $A^+ \in \mathbf{R}^{m_2\times m_1}$ of A as $y^+ = A^+b$. The pseudo-inverse A^+ is the unique matrix in $\mathbf{R}^{m_2\times m_1}$ satisfying the *Moore-Penrose conditions*,

$$AA^+A = A,$$
$$A^+AA^+ = A^+,$$
$$(AA^+)^T = AA^+,$$

and

$$(A^+A)^T = A^+A.$$

If the rank of A is m_2, then $A^+ = (A^TA)^{-1}A^T$. If in addition $m_1 = m_2$, then $A^+ = A^{-1}$.

Our central concern in this chapter is to discuss how a neural network similar to the analog Hopfield neural network of Chapter 6 can be utilized to minimize the function in (7.3), and thereby yield either a solution to the linear system in (7.2), if one exists, or a least-squares approximation thereof, otherwise. As in Chapter 6, the neural network studied in this chapter is analog, and its behavior is described as a function of a real-time parameter $t \geq 0$.

In this chapter, we view the set N of neurons as partitioned into the two sets N_1 and N_2 of sizes m_1 and m_2, respectively (so $n = m_1 + m_2$), and adopt the convention that neurons $n_1, \ldots, n_{m_1}$ belong to N_1, while $n_{m_1+1}, \ldots, n_{m_1+m_2}$ belong to N_2. Our notation in this chapter is entirely analogous to the notation used in Chapter 6. For a neuron $n_i \in N$, $v_i(t)$ is the neuron's *state* at time t and e_i is an *external input* to the neuron. The *synaptic strength* from $n_j \in N$ to $n_i \in N$ is w_{ij}.

The state of a neuron n_i at time $t \geq 0$, in this case an unconstrained real number, is given by

$$\frac{v_i(t)}{R_1} = \sum_{j=1}^{n} w_{ij}v_j(t) + e_i, \tag{7.5}$$

if $n_i \in N_1$, and, if $n_i \in N_2$, by

$$C\frac{dv_i(t)}{dt} = \sum_{j=1}^{n} w_{ij}v_j(t) - \frac{v_i(t)}{R_2}, \tag{7.6}$$

where R_1, C, and R_2 are, respectively, the *resistance* of each neuron in N_1, and the *capacitance* and *resistance* of each neuron in N_2. The right-hand side of (7.5) gives the *input* to neuron $n_i \in N_1$. As in Chapter 6, (7.5) and (7.6) describe the neural network as an electric circuit in which the v_i's are voltages, the e_i's currents, and the w_{ij}'s conductances. We henceforth adopt $R_1 = 1$ and $R_2 = R$ for some $R > 0$.

The values of the neural network's parameters are derived from A and b's elements such that, for all $n_i, n_j \in N$,

$$w_{ij} = \begin{cases} a_{i,j-m_1}, & \text{if } n_i \in N_1 \text{ and } n_j \in N_2; \\ -a_{j,i-m_1}, & \text{if } n_i \in N_2 \text{ and } n_j \in N_1; \\ 0, & \text{otherwise,} \end{cases} \tag{7.7}$$

and

$$e_i = -b_i \tag{7.8}$$

for $n_i \in N_1$ (Figure 7.1). Using (7.7) and (7.8), (7.5) and (7.6) become, respectively,

$$v_i(t) = \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j(t) - b_i \tag{7.9}$$

for $n_i \in N_1$, and

$$C\frac{dv_i(t)}{dt} = -\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(t) - \frac{v_i(t)}{R}, \tag{7.10}$$

for $n_i \in N_2$.

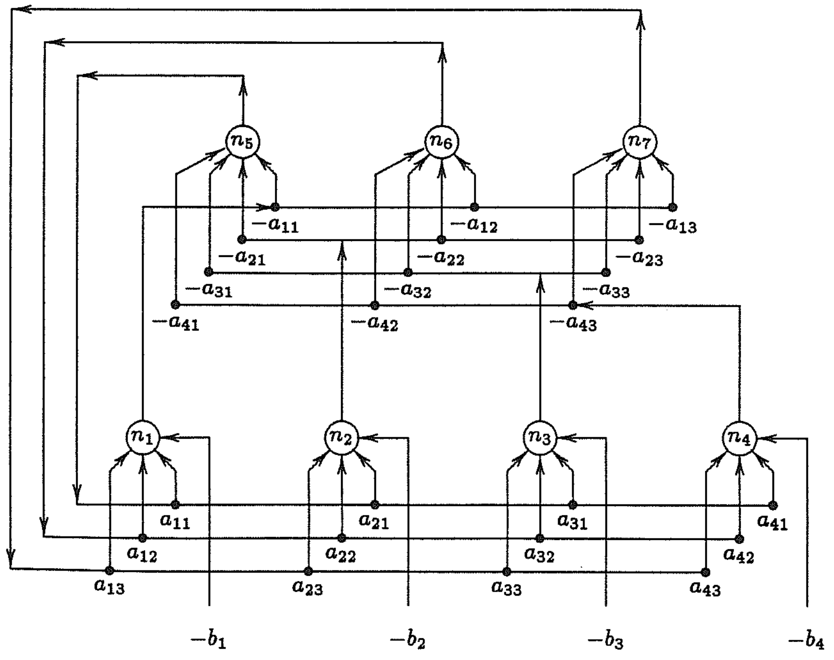


Figure 7.1. *The network for linear system solution is shown here for $m_1 = 4$ and $m_2 = 3$. For the sake of simplicity, synaptic strengths are shown at points at which the output signal of each neuron is tapped to contribute to the input of another neuron.*

Each neuron in N_1 corresponds to an equation in the linear system of (7.2), while each neuron in N_2 corresponds to one of the variables. We now give the theorem that relates the evolution in time of this neural network to a gradient descent on the convex function of (7.3). For $1 \le \ell \le m_2$, let y_ℓ be the variable that corresponds to neuron $n_i \in N_2$. Let also $v(t)$ be the vector in $\mathbf{R}^{m_2}$ whose components are $v_{m_1+1}(t), \ldots, v_{m_1+m_2}(t)$, and $(\nabla\epsilon)_\ell(v(t))$ denote the ℓth component of $\nabla\epsilon(v(t))$.

Theorem 7.1. *For $n_i \in N_2$,*

$$\lim_{R\to\infty} \frac{dv_i(t)}{dt} = -\frac{1}{C}(\nabla\epsilon)_\ell(v(t))$$

for all $t \ge 0$.

Proof: By (7.9) and (7.10), we have

$$\begin{aligned}\frac{dv_i(t)}{dt} &= -\frac{1}{C}\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(t) - \frac{v_i(t)}{R} \\ &= -\frac{1}{C}\sum_{j=1}^{m_1} a_{j,i-m_1}\left(\sum_{k=m_1+1}^{m_1+m_2} a_{j,k-m_1} v_k(t) - b_j\right) - \frac{v_i(t)}{R},\end{aligned}$$

which, in the limit as $R \to \infty$, becomes

$$\frac{dv_i(t)}{dt} = -\frac{1}{C}\sum_{j=1}^{m_1} a_{j,i-m_1}\left(\sum_{k=m_1+1}^{m_1+m_2} a_{j,k-m_1} v_k(t) - b_j\right).$$

By (7.4),

$$(\nabla\epsilon)_\ell(v(t)) = \sum_{j=1}^{m_1} a_{j\ell}\left(\sum_{k=m_1+1}^{m_1+m_2} a_{j,k-m_1} v_k(t) - b_j\right),$$

so the theorem follows by letting $\ell = i - m_1$. ■

By Theorem 7.1, at all times the states of neurons in N_2 are on a path in $\mathbf{R}^{m_2}$ whose tangent at all points is parallel to the gradient of ϵ, and follow this path in the direction opposite to the gradient. What the neural network does is then to perform a gradient descent on ϵ, which as we know must lead to a global minimum. Such global minima correspond to the stable states of the neural network (as in Chapter 6, the *state* of a neural network comprises the states of its neurons).

By (7.4), Theorem 7.1 also implies that the neural network can be regarded as a continuous-time dynamic system of velocity vector field given at time t by

$$\dot{v}(t) = -\frac{1}{C}A^T A v(t) + \frac{1}{C}A^T b,$$

where $\dot{v}(t)$ is the vector in $\mathbf{R}^{m_2}$ whose ℓth component is the first-order derivative with respect to t of $v_\ell(t)$. This correspondence can be used to yield information on the time evolution of the neural network. For example, we know from the literature on dynamic systems that, if the rank of A is m_2, then the rate at which stable points are approached depends solely on the eigenvalues of $A^T A$ and on the value of C, not on the size of the linear system or anything else.

This neural network for linear system solution has many applications, and can also be extended to yield solutions to other related problems. In Section 4.3, we describe one of these extensions, one that requires a little elaboration, to solve linear programming problems. Far simpler extensions are possible, though. For example, consider the problem of computing the inverse of a matrix A for which $m_1 = m_2$. As we know, the inverse A^{-1}, when it exists, is such that

$$AA^{-1} = I,$$

where I is the identity matrix (it has 1's in the main diagonal and 0's elsewhere). If for $1 \le \ell \le m_1$ we let a_ℓ^{-1} denote the ℓth column of A^{-1}, then A^{-1} can be determined by simply solving the m_1 linear systems given by

$$Aa_\ell^{-1} = i_\ell$$

for $1 \le \ell \le m_1$, where i_ℓ is the ℓth column of I. With $2m_1^2$ neurons, A^{-1} can then be determined fully in parallel. If A^{-1} does not exist, then an approximation will be obtained.

7.2. SIMULATION ALGORITHMS

7.2.1. The sequential algorithm

As in Chapter 6, (7.9) and (7.10), together with the initial value $v_i(0)$ of the state of every neuron $n_i \in N$, constitute an initial-value problem, so the recommended approach is to adopt numerical methods of solution. Before we proceed, however, we should note that, unlike the case of analog Hopfield neural networks in Chapter 6, we must now approach the selection of initial states carefully. Specifically, neurons in N_2 (those that correspond to variables of the linear system) may have their initial states selected arbitrarily. Neurons in N_1 (corresponding to equations in the linear system), on the other hand, must have their initial states chosen so that they are consistent with those of neurons in N_2 by (7.9). This is so because what the neurons in N_1 do is to apply a linear function on the states of neurons in N_2, so their states must at all times reflect the value of this function.

The numerical approaches in this chapter are the same employed in Chapter 6, namely Euler's method and the Runge-Kutta method. We have discussed these methods in some detail in Section 6.2.1, and shall for this reason in this chapter only indicate how they apply to the network that solves linear systems. The application of Euler's method is derived directly from (6.6), (7.9), and (7.10), but taking into

account that, by Theorem 7.1, the resistance R is required to have a very high value. Euler's method then yields

$$v_i(\tau) = \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j(\tau - \Delta t) - b_i \tag{7.11}$$

for all $n_i \in N_1$, and

$$v_i(\tau) = v_i(\tau - \Delta t) + \frac{\Delta t}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(\tau - \Delta t)\right) \tag{7.12}$$

for all $n_i \in N_2$, at the discrete times $\tau \in \{\Delta t, 2\Delta t, \ldots\}$.

Similarly, a direct application of (6.8) and (7.10) yields

$$v_i(\tau) = v_i(\tau - \Delta t) + \frac{\Delta t}{6}(K_1^i + 2K_2^i + 2K_3^i + K_4^i), \tag{7.13}$$

where, by (6.9),

$$\begin{aligned}
K_1^i &= \frac{1}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(\tau - \Delta t)\right) \\
K_2^i &= \frac{1}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1}\left(v_j(\tau - \Delta t) + \frac{K_1^j}{2}\right)\right) \\
K_3^i &= \frac{1}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1}\left(v_j(\tau - \Delta t) + \frac{K_2^j}{2}\right)\right) \\
K_4^i &= \frac{1}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1}\left(v_j(\tau - \Delta t) + K_3^j\right)\right),
\end{aligned} \tag{7.14}$$

for all $n_i \in N_2$ at the discrete times $\tau \in \{\Delta t, 2\Delta t, \ldots\}$. (7.13) and (7.14) are, in conjunction with (7.11), how to apply the Runge-Kutta method to solve the linear-system neural network.

Both the approach based on Euler's method and the one based on the Runge-Kutta method allow us to view the neural network as an FC automaton network in which G has edges between every node in N_1 and every node in N_2, and only these. Letting $s = \tau/\Delta t$ yields an updating function f such that $x_i(0) = v_i(0)$ and $x_i(s) = v_i(\tau)$ for all $n_i \in N$ and all $s > 0$ (cf. Chapter 1). This updating function is either based on (7.11) and (7.12), if Euler's method is applied, or on (7.11), (7.13), and (7.14), in the case of the Runge-Kutta method. This FC automaton network can be simulated at discretized values of t by the sequential algorithm given next as Algorithm *Seq_Least_Squares*. This algorithm is entirely similar to Algorithm *Seq_Analog_Hopfield* of Section 6.2.1, and runs until no neuron in N_2 has to be updated any further, which happens when the states of all neurons in N_2 are within a tolerance $\varepsilon > 0$ of their previous values. Algorithm *Seq_Least_Squares* is based on Euler's method, so it employs (7.11) and (7.12), but it should be immediate to note that the Runge-Kutta method can be used equally easily.

```
Algorithm Seq_Least_Squares:
    $\tau := 0$;
    repeat
        $\tau := \tau + \Delta t$;
        for $i := 1$ to $m_1$ do
            $v_i(\tau) := \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j(\tau - \Delta t) - b_i$;
        for $i := m_1 + 1$ to $m_1 + m_2$ do
            $v_i(\tau) := v_i(\tau - \Delta t) + \frac{\Delta t}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(\tau - \Delta t)\right)$
    until $|v_i(\tau) - v_i(\tau - \Delta t)| \le \varepsilon$ for all $n_i \in N_2$.
```

7.2.2. The distributed parallel algorithm

When the neural network for linear system solution has its equations solved by Euler's method or the Runge-Kutta method, it can, as we discussed in Section 7.2.1, be regarded as an FC automaton network. The reason for this is that these methods, as others, work by approximating the solutions at discrete times, and at each time employs information about the previous instant only.

Viewed as an FC automaton network, the neural network for linear system solution can be simulated by a distributed parallel algorithm derived from the template given in Section 4.3.2 for the distributed parallel simulation of FC automaton networks in general. Processor p_0 is in this case a convergence detector, and utilizes the tolerance ε discussed in Section 7.2.1. In order to completely specify the simulation algorithm, we must describe the procedures INITIALIZE$_i$, UPDATE$_i$(*MSG*$_i$), and NOTIFY$_i$ for every processor p_i such that $n_i \in N$. As described in Section 4.3.2, *MSG*$_i$ contains exactly one message from each of p_i's neighbors. These messages are the states of the corresponding neurons and are needed to update n_i's state.

The three procedures we give are based on (7.11) and (7.12), as Algorithm *Seq_Least_Squares*. From our discussion in Section 7.2.1, it should nevertheless be clear that they might as well be based on (7.11), (7.13), and (7.14) just as easily.

```
INITIALIZE_i:
    1. Let $v_i := v_i(0)$;
    2. Send $v_i$ to $p_0$;
    3. Send $v_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
UPDATE_i(MSG_i):
    1. Let $v_j \in MSG_i$ be the state of $n_j$ for all $n_j \in \mathcal{N}(n_i)$;
    2. Let
            $v_i := \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j - b_i$,
        if $n_i \in N_1$, or
            $v_i := v_i + \frac{\Delta t}{C}\left(-\sum_{j=1}^{m_1} a_{j,i-m_1} v_j\right)$,
        if $n_i \in N_2$.
NOTIFY_i:
    1. Send $v_i$ to $p_0$;
    2. Send $v_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

In these procedures, v_i is a variable local to p_i.

7.3. LINEAR PROGRAMMING

As we noted in Section 7.1, various extensions of the neural network to solve linear systems are possible. In this section, we describe an extension to solve linear programming problems. A *linear programming problem*, or *linear program*, is an optimization problem whose objective function is a linear function and whose constraints are also given by linear equalities or inequalities.

A linear program is said to be in the *standard form* when it is posed as

$$\begin{aligned} &\text{minimize } c^T y \\ &\text{subject to } Ay = b \\ &\qquad y \ge 0, \end{aligned} \tag{7.15}$$

where $A \in \mathbf{R}^{m_1 \times m_2}$, $b \in \mathbf{R}^{m_1}$, and $c, y \in \mathbf{R}^{m_2}$. The *dual* of the linear program in (7.15) is the linear program

$$\begin{aligned} &\text{maximize } b^T \pi \\ &\text{subject to } A^T \pi \le c, \end{aligned} \tag{7.16}$$

where $\pi \in \mathbf{R}^{m_1}$ (the linear program in (7.15) is also referred to as the *primal*). The linear programs in (7.15) and (7.16) are related to each other by the following remarkable property. If $y^* \in \mathbf{R}^{m_2}$ and $\pi^* \in \mathbf{R}^{m_1}$ are optima respectively of the primal and the dual, then

$$c^T y^* = b^T \pi^*. \tag{7.17}$$

Our neural network approach to solve the linear program in (7.15) has the following general appearance. First we set up two neural networks to handle the "linear-system" constraints of the primal and the dual. Then we connect the two networks together by means of an additional equation given by the primal-dual optimality equation of (7.17). Of course, the first phase cannot be so straightforward, as the constraints in the primal and dual problems are not exactly linear systems.

We start with a neural network to handle the constraints of the dual problem, which is simpler. The main reason why we cannot employ the neural network for linear system solution directly are the inequalities appearing in $A^T\pi \leq c$. Furthermore, the system matrix is now A^T. The only modifications we need are then to interchange the roles of N_1 and N_2 (although neurons in the new N_1 are still numbered first), and to introduce a slight nonlinearity in the equations describing the behavior of neurons in the new N_2. Specifically, for this network we replace (7.5) with

$$v_i(t) = \min\left\{0, \sum_{j=1}^{n} w_{ij} v_j(t) + e_i\right\} \tag{7.18}$$

for all $n_i \in N_2$ (Figure 7.2). The behavior of the network is then given by

$$v_i(t) = \min\left\{0, \sum_{j=1}^{m_1} a_{j,i-m_1} v_j(t) - c_{i-m_1}\right\} \tag{7.19}$$

for $n_i \in N_2$, and by

$$C\frac{dv_i(t)}{dt} = -\sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j(t) - \frac{v_i(t)}{R} \tag{7.20}$$

for $n_i \in N_1$, following (7.5) and (7.6), respectively (note that the system matrix is now A^T, so N_1 and N_2 are the new ones).

The neural network to handle the constraints of the primal problem must also be adapted, this time because of the requirement that y have nonnegative entries in (7.15). In order to handle this, we introduce appropriate nonlinearities in the behavior of the neurons in N_2. Specifically, (7.6) becomes

$$v_i(t) = \max\{0, u_i(t)\} \tag{7.21}$$

for all $n_i \in N_2$ (Figure 7.2), where $u_i(t)$ is the *potential* of n_i, similar to the potential of a neuron in an analog Hopfield neural network (cf. Chapter 6), given as in (7.6) by

$$C\frac{du_i(t)}{dt} = \sum_{j=1}^{n} w_{ij} v_j(t) - \frac{u_i(t)}{R}. \tag{7.22}$$

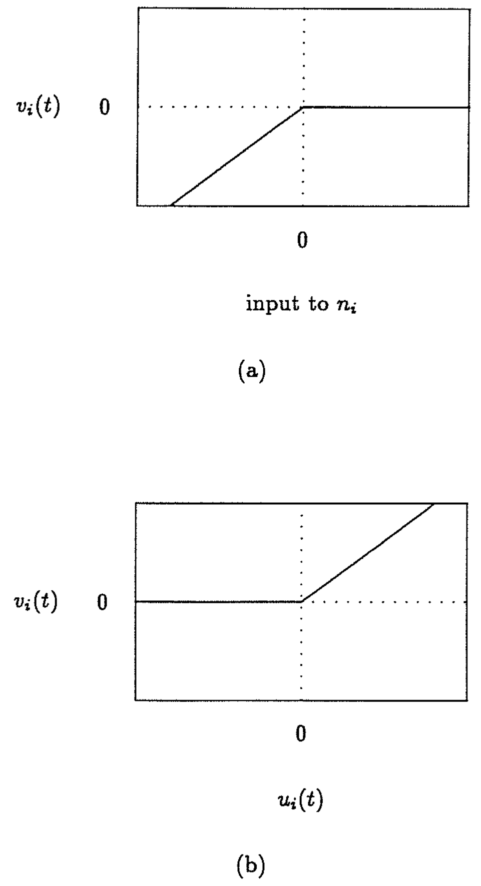


Figure 7.2. *When the neural network to solve linear systems is adapted to solve linear programs, some of the neurons must be endowed with nonlinear characteristics. A neuron in (the new) N_2 dealing with the dual formulation of the problem must have a 0 state whenever its input is positive (a), while a neuron in N_2 dealing with the primal formulation of the problem must have a 0 state whenever its potential is negative (b). Their behavior is otherwise linear.*

The behavior of the neural network is then given by

$$v_i(t) = \sum_{j=m_1+1}^{m_1+m_2} a_{i,j-m_1} v_j(t) - b_i \tag{7.23}$$

for all $n_i \in N_1$, and by (7.21), where $u_i(t)$ is such that

$$C\frac{du_i(t)}{dt} = -\sum_{j=1}^{m_1} a_{j,i-m_1} v_j(t) - \frac{u_i(t)}{R}, \tag{7.24}$$

for all $n_i \in N_2$, according to (7.5) and (7.22).

These two neural networks are then used to build a single neural network, with an additional neuron to represent the equation $c^T y - b^T \pi = 0$ of primal-dual optimality from (7.17). The resulting network is, aside from the nonlinearities of some of its neurons, viewed as solving the $(m_1 + m_2 + 1) \times (m_1 + m_2)$ linear system

$$\begin{pmatrix} A & 0 \\ 0 & A^T \\ c^T & -b^T \end{pmatrix} \begin{pmatrix} y \\ \pi \end{pmatrix} = \begin{pmatrix} b \\ c \\ 0 \end{pmatrix}.$$

With very high values for the resistance R, as required by Theorem 7.1, (7.19), (7.20), (7.21), (7.23), and (7.24) can then be used to solve the linear program in (7.15).

7.4. BIBLIOGRAPHIC NOTES

The neural networks treated in this chapter are, as the analog Hopfield neural networks of Chapter 6, in the class of analog neural networks considered by Mead (1989). A reference on linear systems and the least-squares problem to complement the material given in Section 7.1 is Golub and van Loan (1983). The neural network approach to the solution of linear systems is from de Carvalho and Barbosa (1992). Other approaches to this and related problems emphasizing the distributed parallel character of the methods are given by Bertsekas and Tsitsiklis (1989). General literature on the gradient-descent method and on dynamic systems includes, respectively, Luenberger (1973) and Luenberger (1979).

The reader is referred to Wylie (1975) and Braun (1978) for details on Euler's method and the Runge-Kutta method, employed in Section 7.2 to solve the differential equations describing the behavior of the network.

Reference texts on the traditional approaches to linear programming are Dantzig (1963), Luenberger (1973), and Papadimitriou and Steiglitz (1982). The ellipsoid method, which established the polynomial-time complexity of linear programming, was introduced by Khachiyan (1979), and utilized by Grötschel, Lovász, and Schrijver (1981) in the derivation of interesting results in combinatorial optimization. Practical methods of polynomial-time complexity have appeared in the

wake of the paper by Karmarkar (1984). These are the so-called interior-point or feasible-direction methods, and are surveyed by Gonzaga (1992). The neural-network solution we discuss in Section 7.3 is based on de Carvalho and Barbosa (1992). Other solutions based on neural networks were given by Hopfield (1986) and Barbosa and de Carvalho (1990), the latter based on a feasible-direction algorithm by Herskovits (1986).

Part 4

Partially concurrent automaton networks

In this part of the book, we treat PC automaton networks. Similarly to the chapters of Part 3, the overall approach is to present various PC automaton network models, along with their most relevant properties, and to discuss the most prominent application areas of each model. Algorithms for the sequential and distributed parallel simulation of each model are also provided.

Part 4 contains Chapters 8 through 10. In Chapter 8, the topic of binary Hopfield neural networks is treated. Chapters 9 and 10 are relatively closely related, treating respectively Markov random fields and Bayesian networks, which can in many senses be regarded as instances of the former.

8

Binary Hopfield neural networks

In this chapter, we discuss binary Hopfield neural networks, which are introduced, along with their essential properties, in Section 8.1. These networks are characterized as PC automaton networks in Section 8.2, where simulation algorithms are presented. The use of binary Hopfield neural networks as associative memories is treated in Section 8.3. In Section 8.4, they are used as a tool for combinatorial optimization to solve the minimum node cover problem. Bibliographic notes follow in Section 8.5.

8.1. THE MODEL AND BASIC PROPERTIES

As in Chapters 6 and 7, in this chapter the set N is once again viewed as a set of *artificial neurons* (or simply *neurons*). In a binary Hopfield neural network, the *state* v_i of a neuron n_i is restricted to the set $\{0, 1\}$, and is given by

$$v_i := \text{step}\left(\sum_{j=1}^{n} w_{ij}v_j + e_i - \theta_i\right), \tag{8.1}$$

where

$$\text{step}(y) = \begin{cases} 0, & \text{if } y \leq 0; \\ 1, & \text{if } y > 0, \end{cases}$$

as illustrated in Figure 8.1. (The use of an assignment (:=) instead of an equality (=) in (8.1) is purposeful: equality can only hold under stable conditions, as we shall discuss shortly.) The remainder of the notation in (8.1) is entirely analogous to the one used in Chapters 6 and 7: w_{ij} is the *synaptic strength* from neuron n_j to neuron n_i and e_i is an *external input* to n_i. The *potential* of n_i is in the binary case given directly by the *input* to the neuron, i.e., by

$$\sum_{j=1}^{n} w_{ij}v_j + e_i,$$

and θ_i is its *threshold potential* (or simply *threshold*). Despite this similarity in notation, it should be noted, still in comparison with Chapter 6, that a neuron's state is no longer described as a function of a real-time parameter t, which in the context of the models discussed in Chapters 8 through 10 is meaningless.

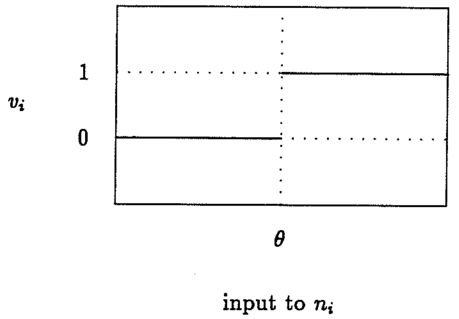


Figure 8.1. *The state v_i of a neuron in a binary Hopfield neural network is* 1 *if the input to the neuron is strictly larger than θ_i, and is* 0 *for inputs no larger than θ_i.*

As anticipated by Theorem 6.2, the binary Hopfield neural network can be regarded as the limiting case of the analog Hopfield neural network of Chapter 6 when every neuron's gain becomes very large. In such limiting situations, the derivatives that appear in (6.2) are no longer needed, and a neuron's state is given directly as a function of its input by (8.1), which in turn can be viewed as the high-gain analogue of (6.1).

Just as in the analog case, the collective behavior of the interconnected neurons is in general hard to analyze. Once more, however, an *energy* function can be defined which evidences very interesting properties of the network's dynamic behavior. Specifically, let

$$E = -\sum_{i=1}^{n}\sum_{j=i}^{n} w_{ij}v_iv_j - \sum_{i=1}^{n} e_iv_i + \sum_{i=1}^{n}\theta_iv_i. \tag{8.2}$$

A comparison of (8.2) with (6.4) reveals that the energy function is now identical to the energy of the analog Hopfield neural network when gains are very high and resistances are unit. Not surprisingly, then, Theorem 8.1 given next can be thought of as the binary counterpart of Theorem 6.1.

Theorem 8.1. *Suppose that $w_{ij} = w_{ji}$ for all $n_i, n_j \in N$, and let $\mathcal{K} \subseteq N$ be such that, if $n_i, n_j \in \mathcal{K}$, then $w_{ij} = 0$. If E_1 and E_2 are, respectively, the values of E immediately before and after the neurons $n_i \in \mathcal{K}$ are updated concurrently as in (8.1), then $E_2 - E_1 \leq 0$.*

Proof: By (8.2), and because $w_{ij} = w_{ji}$ for all $n_i, n_j \in N$ and $w_{ij} = 0$ for all $n_i, n_j \in \mathcal{K}$, we have

$$E_1 = \sum_{n_i\in\mathcal{K}} v_i\left(-\sum_{n_j\notin\mathcal{K}} w_{ij}v_j - e_i + \theta_i\right) + \sum_{n_i\notin\mathcal{K}} v_i\left(-\frac{1}{2}\sum_{n_j\notin\mathcal{K}} w_{ij}v_j - e_i + \theta_i\right)$$

and

$$E_2 = \sum_{n_i\in\mathcal{K}} \bar{v}_i\left(-\sum_{n_j\notin\mathcal{K}} w_{ij}v_j - e_i + \theta_i\right) + \sum_{n_i\notin\mathcal{K}} v_i\left(-\frac{1}{2}\sum_{n_j\notin\mathcal{K}} w_{ij}v_j - e_i + \theta_i\right),$$

where $\bar{v}_i$ is the result of applying (8.1) to $n_i \in \mathcal{K}$ when its state is v_i. As a consequence,

$$E_2 - E_1 = \sum_{n_i\in\mathcal{K}} (v_i - \bar{v}_i)\left(\sum_{n_j\notin\mathcal{K}} w_{ij}v_j + e_i - \theta_i\right),$$

which is nonpositive, as, by (8.1),

$$\bar{v}_i = \text{step}\left(\sum_{j=1}^{n} w_{ij}v_j + e_i - \theta_i\right)$$

for all $n_i \in \mathcal{K}$. ■

Theorem 8.1 should be examined with care. As Theorem 6.1 of the analog case, it too states that the network's energy, as given by (8.2), does not increase as the neurons are updated (cf. Figure 6.2). There is, however, one fundamental difference, as this behavior is guaranteed to occur if no two neurons interconnected by a nonzero synaptic strength have their states updated concurrently. This is also in fact an "only if" condition, as can be verified by the following simple example. Let $n = 3$, $w_{12} = w_{21} = 1$, $w_{13} = w_{31} = 1$, and $w_{23} = w_{32} = -1.5$. Suppose also that $e_i = \theta_i = 0$ for $i = 1, 2, 3$. If the neuron states are $v_1 = 1$, $v_2 = 1$, and $v_3 = 1$, then the energy goes from $E = -0.5$ to $E = 0$ if the three neurons have their states updated concurrently, whereas it goes from $E = -0.5$ to $E = -1.0$ if the updates are done, for example, in the order n_1, then n_2, then n_3. In the former case, the energy increase is due to the fact that neurons interconnected by nonzero synaptic strengths have their states updated concurrently.

As in the case of analog Hopfield neural networks, Theorem 8.1 implies that the evolution of the binary network according to the rule in (8.1) leads to local minima of the energy function of (8.2), as long as no neuron is indefinitely precluded from having its state updated. This energy is then a Lyapunov function of the dynamic system that the network represents, as the system's stable states are minima of the energy. Such minima are referred to as the network's *stable states* (a *state* of the network is here, as in previous chapters, given by the assembled states of the individual neurons). Note that only at stable states does (8.1) hold with equality (instead of assignment).

The continual minimization of the energy is central to the applications of the binary Hopfield neural network, as in models of associative memory and in optimization. Both types of application can, of course, be tackled by the analog model as well. In fact, one important problem in combinatorial optimization, TSP, was discussed in Section 6.3 as an application of that model. Later in this chapter, in Sections 8.3 and 8.4, we shall discuss, respectively, how the binary network can be used to model an associative memory, and solutions to the minimum node cover problem. Discussing the latter problem within the framework of the binary model is especially important in view of the material presented in Chapter 9, where we discuss how to escape from local minima (to seek global minima) by means of stochastic decisions embedded in the binary model.

8.2. SIMULATION ALGORITHMS

8.2.1. The sequential algorithm

The models discussed in Chapters 6 and 7 had their behavior described by differential equations that governed their evolution in real time. When the solution of these differential equations was approached via a finite-difference method, the models' characterization as FC automaton networks emerged, as the variations of this method we discussed (Euler's method and the Runge-Kutta method) solve the differential equations at discrete time instants by employing at each instant information concerning the previous instant only.

Binary Hopfield neural networks, on the other hand, behave according to a simple "threshold rule," expressed as in (8.1) by a step function around the threshold potential θ_i for all $n_i \in N$. A real-time parameter has then no place in describing the network's behavior, which depends essentially on the relative concurrency and frequency with which neurons have their states updated. The dependency on the relative concurrency of updates is given by Theorem 8.1, according to which a sufficient condition for the updates of neuron states to lead to a nonincreasing succession of values of the energy function in (8.2) is that no two neurons be allowed to have their states updated concurrently if they are interconnected by a nonzero synaptic strength. As we also remarked in Section 8.1, this condition is in addition necessary, as it is rather easy to come up with examples in which the energy increases with respect to its previous value if the condition is not met.

The dynamic behavior of a binary Hopfield neural network depends also on the relative frequency with which neurons have their states updated. As we briefly remarked in Section 8.1, it is a consequence of the assertion of Theorem 8.1 that local minima of the energy function may be reached as the neurons have their

states updated. This is only guaranteed to happen, however, if every neuron is "sometimes" given a chance to have its state updated. For a simple example of a situation in which a local minimum may not be reached, consider a neuron of zero threshold potential and suppose that it is connected to other neurons exclusively by negative synaptic strengths. If this neuron starts off with its state set to 1 and is never given a chance to have it updated, then evidently no local minimum of the energy will ever be reached, as in such a local minimum the state of that neuron must be 0. As in most applications of the binary Hopfield neural network the energy's minima play an important role, there has to be a way of ensuring that every neuron is "occasionally" given a chance to have its state updated.

These two aspects of the behavior of a binary Hopfield neural network, namely the relative concurrency and frequency with which neuron states are updated, lead to a characterization of such networks as PC automaton networks. Such a PC automaton network is obtained by letting G's edges correspond to pairs $n_i, n_j \in N$ such that $w_{ij} \neq 0$, inasmuch as in G the set $\mathcal{K}$ alluded to in Theorem 8.1 is an independent set. The updating function f in this case is such that $x_i(s)$ is given by the initial value of v_i for $s = 0$, and for $s > 0$ by the value of v_i obtained after the most recent pulse in which n_i was in the independent set. We shall denote the initial value of v_i by v_i^0. This PC automaton network can be simulated by a sequential algorithm that scans the neurons in a fixed order and updates their states according to (8.1). Such an algorithm is given next. It runs until the updated state of every neuron is the same as the neuron's previous state (denoted, in the algorithm, by v_i^- for $n_i \in N$) after a full scan.

Algorithm *Seq_Binary_Hopfield*:

```
for $i := 1$ to $n$ do
    $v_i := v_i^0$;
repeat
    for $i := 1$ to $n$ do
        begin
            $v_i^- := v_i$;
            $v_i := \text{step}\left(\sum_{j=1}^{n} w_{ij} v_j + e_i - \theta_i\right)$
        end
until $v_i = v_i^-$ for all $n_i \in N$.
```

Algorithm *Seq_Binary_Hopfield* is fully deterministic. Given a set of initial neuron states, the algorithm produces the exact same sequence of neuron states and leads the network to the exact same stable state whenever it is started at that set. This can be very undesirable in certain applications. For example, if the network is used as a model of associative memory, then it is desirable that, when started at a state approximately equally "close" to more than one local minimum of the energy, the network will at times evolve toward one of these minima, at other times toward another. A simple modification to Algorithm *Seq_Binary_Hopfield* can produce this effect on a probabilistic basis. Whenever in the algorithm it is a neuron's turn to have its state updated, do it with a pre-established fixed probability, otherwise pass. In the long run, the probability that a neuron is never updated is nearly zero. Also, if this probability is the same for all neurons, then in the long run every neuron has its state updated approximately equally frequently. The control for termination of the algorithm is in this case a little more complex, as the test of whether $v_i = v_i^-$ can only be performed after n_i has had a chance to modify v_i with respect to the value it had when it was assigned to v_i^-.

8.2.2. The distributed parallel algorithm

As we discussed in Section 8.2.1, a binary Hopfield neural network can be viewed as a PC automaton network, inasmuch as it can only be ensured to promote a minimization of the energy function if no two neurons have their states updated concurrently if they are interconnected by a nonzero synaptic strength, and furthermore no neuron is indefinitely precluded from having its state updated.

This established, the distributed parallel simulation of a binary Hopfield neural network can be achieved according to the template given in Section 4.3.3. For this type of network, processor p_0 participates as a detector of the convergence of the simulation, which in this case is determined based on the fact that all further updates would have no effect on the neurons' states. The template is filled by specifying the procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for every processor p_i such that $n_i \in N$. We know from Section 4.3.3 that MSG_i contains exactly one message from each of p_i's neighbors. In the case of the binary Hopfield neural network, each of these messages is the state of the neuron that corresponds to that neighbor, to be used in updating n_i's state.

The main steps of the procedures to fill the template are given next. They are, as Algorithm *Seq_Binary_Hopfield*, based on (8.1) for the updating of the neurons' states.

```
INITIALIZE_i:
  1. Let $v_i := v_i^0$;
  2. Send $v_i$ to $p_0$;
  3. Send $v_i$ to every downstream neighbor $p_j$ of $p_i$.
UPDATE_i($MSG_i$):
  1. Let $v_j \in MSG_i$ be the state of $n_j$ for all $n_j \in \mathcal{N}(n_i)$;
  2. Let
         $v_i := \mathrm{step}\left(\sum_{j=1}^{n} w_{ij}v_j + e_i - \theta_i\right)$.
NOTIFY_i:
  1. Send $v_i$ to $p_0$;
  2. Send $v_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

The simulation these procedures conduct is, just as in the case of Algorithm *Seq_Binary_Hopfield*, fully deterministic. However, our comments in Section 8.2.1 concerning the introduction of some nondeterminism by means of probabilistic decisions apply equally to the distributed parallel case.

8.3. THE NETWORK AS AN ASSOCIATIVE MEMORY

An *associative memory*, also called a *content-addressable memory*, is a device that allows the storage of items for later retrieval based on incomplete information about the item to be retrieved. Associative memories have numerous applications, and in these applications the items to be stored are typically partitioned into fields, each with a special semantics. One or more of these fields are the key to the item, and are presented to the memory when the item needs to be retrieved. Retrieval of the item then allows the remaining fields to be available for use.

The binary Hopfield neural network discussed in Section 8.1 can be used as a model of associative memory. The basic idea is quite simple: as the network's evolution leads to stable states, each item to be stored can be associated with one of these states; when started at a state "close" enough to the stable state representing the desired item, the network then evolves to that stable state and the item can be fully retrieved. Setting up a binary Hopfield neural network to function as an associative memory involves at least three important issues, namely a measure for the "closeness" of states, an approach to the determination of the network's parameters (chiefly synaptic strengths), and an assessment of the number of items that can be stored in the network (probably as a function of its number of neurons) so that items do not interfere with one another so badly that none can ever be successfully retrieved.

In our discussion throughout this section (and only in this section), we shall change our convention regarding a binary Hopfield neural network, and assume that the state v_i of $n_i \in N$ is in $\{-1, 1\}$, rather than in $\{0, 1\}$. Consequently, (8.1) becomes

$$v_i := \mathrm{sign}\left(\sum_{j=1}^{n} w_{ij}v_j + e_i - \theta_i\right), \tag{8.3}$$

where

$$\mathrm{sign}(y) = \begin{cases} -1, & \text{if } y \le 0; \\ 1, & \text{if } y > 0. \end{cases}$$

The change in the name of the function in (8.3) reflects the fact that it now gives, in the form of -1 or 1, the sign (negative or positive, respectively) of its argument. This is done for notational convenience only, and does not change any of the properties discussed in Section 8.1. In particular, Theorem 8.1 still holds as is. Another assumption throughout this section will be that $e_i = \theta_i = 0$ for all $n_i \in N$.

An item to be stored in the associative memory will henceforth be referred to as a *pattern*. Within the context of viewing a binary Hopfield neural network as an associative memory, the usual measure of "closeness" between two network states is the *Hamming distance*, defined as the number of neurons whose states are different in the two network states, that is, as

$$\frac{1}{4}\sum_{i=1}^{n}\left[(1+v_i^1)(1-v_i^2)+(1-v_i^1)(1+v_i^2)\right],$$

where $v_1^1, \ldots, v_n^1$ and $v_1^2, \ldots, v_n^2$ are the two network states.

In order to gain some intuition on how to set up a network's synaptic strengths so that it functions as an associative memory, let us first consider the simple case in which one single pattern $\mu \in \{-1,1\}^n$, of components $\mu_1, \ldots, \mu_n$, is to be stored. This pattern is stable if and only if, following (8.3),

$$\mu_i = \operatorname{sign}\left(\sum_{j=1}^{n} w_{ij}\mu_j\right) \tag{8.4}$$

for all $n_i \in N$. Suppose we adopt synaptic strengths with values

$$w_{ij} = \begin{cases} \frac{1}{n}\mu_i\mu_j, & \text{if } i \neq j; \\ 0, & \text{if } i = j \end{cases} \tag{8.5}$$

for all $n_i, n_j \in N$. (8.5) then yields, for all $n_i \in N$,

$$\sum_{j=1}^{n} w_{ij}\mu_j = \frac{1}{n}\mu_i \sum_{\substack{j=1 \\ j\neq i}}^{n} \mu_j\mu_j \tag{8.6}$$

as the input to n_i. The summation in the right-hand side of (8.6) is always positive, so

$$\sum_{j=1}^{n} w_{ij}\mu_j$$

has the sign of μ_i, thereby satisfying (8.4), and then μ is stable.

What is even more interesting about the synaptic strength assignment of (8.5) is that the single pattern μ is the stable state to which the state of the network converges if not too far from μ initially. For let $\nu \in \{-1,1\}^n$ be the initial network state, and suppose that the Hamming distance between μ and ν is at most $\lceil (n-2)/2 \rceil$. Under these circumstances, the input to neuron n_i is, for all $n_i \in N$, given by

$$\sum_{j=1}^{n} w_{ij}\nu_j = \frac{1}{n}\mu_i\left(\sum_{\substack{j=1 \\ j\neq i \\ \mu_i=\nu_i}}^{n} \mu_j\nu_j + \sum_{\substack{j=1 \\ j\neq i \\ \mu_i\neq\nu_i}}^{n} \mu_j\nu_j\right). \tag{8.7}$$

The two summations in the right-hand side of (8.7) are, respectively, the number of neurons at which μ and ν agree and minus the number of neurons at which they differ. The overall summation is then, by hypothesis, strictly positive, so

$$\sum_{j=1}^{n} w_{ij}\nu_j$$

has the sign of μ_i for all $n_i \in N$. As a result, neurons at which μ and ν agree do not change states, whereas those at which μ and ν differ have their states altered. Pattern μ is then fully recovered. Had we started out with a state ν differing from μ at no less than $\lfloor (n+2)/2 \rfloor$ neurons, then by (8.7) all neurons at which ν and μ agree would change their states, and consequently the network would settle at pattern $-\mu$, which is stable by the exact same argument we used to argue for the stability of μ (Figure 8.2). If, for n even, ν agreed with μ at exactly $n/2$ neurons, then the network would settle at one of μ of $-\mu$, as the first neuron to be updated would imbalance the state of the network so that it would have at most $(n-2)/2$ or at least $(n+2)/2$ neurons agreeing with μ.

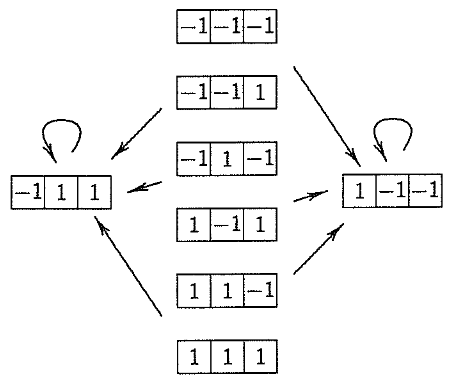


Figure 8.2. *Patterns $\mu = (-1,1,1)$ and $-\mu = (1,-1,-1)$ are both stable, given the appropriate choice of synaptic strengths. When the network is started at any of the six other patterns, either μ or $-\mu$ is recovered, depending on which one is closest to the starting pattern by the Hamming distance.*

The case of more than one pattern is more complex, and does not yield so easily to a precise analytical treatment. If we have the P patterns $\mu^1, \ldots, \mu^P \in \{-1,1\}^n$ for some $P \geq 1$, then a generalization of (8.5) yields the synaptic strengths

$$w_{ij} = \begin{cases} \frac{1}{n}\sum_{p=1}^{P} \mu_i^p\mu_j^p, & \text{if } i \neq j; \\ 0, & \text{if } i = j \end{cases} \tag{8.8}$$

for all $n_i, n_j \in N$. (8.8) is known as the *Hebb rule* for learning, as setting up the synaptic strengths can be viewed as the process whereby the network learns the patterns to be stored. For a pattern μ^q for some q such that $1 \leq q \leq P$, the input to n_i is, by (8.8),

$$\begin{aligned}\sum_{j=1}^{n} w_{ij}\mu_j^q &= \frac{1}{n}\sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{p=1}^{P}\mu_i^p\mu_j^p\mu_j^q \\ &= \frac{n-1}{n}\mu_i^q + \sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{\substack{p=1\\ p\neq q}}^{P}\frac{\mu_i^p\mu_j^p\mu_j^q}{n},\end{aligned} \tag{8.9}$$

for all $n_i \in N$. Now consider the double summation in the second line of (8.9), often referred to as the *crosstalk term*. If this summation is strictly less than $(n-1)/n$ for all $n_i \in N$, then the input to each n_i has the sign of μ_i^q, so μ^q is stable, as the condition for stability of μ^q reads, by (8.4),

$$\mu_i^q = \text{sign}\left(\sum_{j=1}^{n} w_{ij}\mu_j^q\right), \tag{8.10}$$

for all $n_i \in N$, with the w_{ij}'s given as in (8.8).

If the crosstalk term is larger than or equal to $(n-1)/n$ for some pattern μ^q at some neuron n_i, then μ^q is unstable. Surely, we expect more patterns to be unstable as P increases, i.e., as we attempt to store more patterns in the network. One fundamental issue is then the assessment of the *capacity* of the network, that is, of the maximum number of patterns the network can store. Put this way, the concept of a network's capacity is of course imprecise, as it depends intimately on the desired accuracy in the process of recovering patterns. Various analyses have been carried out in the literature to determine the capacity of a binary Hopfield neural network. A very brief account of their conclusions under the assumption that the patterns to be stored are generated randomly by picking either -1 or 1 with equal probability at all neurons is the following. If errors in a small fraction of the neurons can be tolerated, then the capacity is proportional to n, but not larger than $0.138n$ (in this case, about 0.36% of the neurons are incorrect when the network stabilizes). If, on the other hand, all (or practically all) neurons are required to be retrieved perfectly, then the network's capacity is proportional to $n/\log n$.

When a binary Hopfield neural network is employed as an associative memory, the graph G that gives the structure of the corresponding PC automaton network depends intimately on the patterns to be stored. Suppose, for example, that we take the synaptic strengths to be given as in (8.8). If patterns are generated randomly as discussed previously, then the probability that a synaptic strength is equal to zero when P is even (no synaptic strength can be zero if P is odd) can be evaluated by considering a sequence of P independent trials in which one of the pairs $(-1,-1)$, $(-1,1)$, $(1,-1)$, or $(1,1)$ is chosen with probability $1/4$ at each trial (such identical, independent trials are known as *Bernoulli trials*). The probability that in this

sequence the number of pairs with members of the same sign is equal to the number of pairs with members of different signs is

$$\left(\frac{1}{4}+\frac{1}{4}\right)^{P/2}\left(\frac{1}{4}+\frac{1}{4}\right)^{P/2}.$$

Considering the variety of sequences in which this situation can occur, we then have that the probability of a zero synaptic strength is given by

$$\binom{P}{P/2}\left(\frac{1}{2}\right)^{P} \tag{8.11}$$

if P is even, and by 0 if P is odd. For $P = 138$ (the largest number of patterns, according to one of the criteria in our previous discussion, if $n = 1000$), (8.11) yields a probability approximately equal to 0.07, although the average synaptic strength can be shown, analogously to the reasoning that led to (8.11), to be zero. For larger values of P, this probability is even smaller, so G is a very dense graph and the distributed parallel simulation of the PC automaton network yields very little concurrency. Synaptic strengths generated by real applications, on the other hand, may behave differently, and may then yield a reasonable amount of concurrency during the distributed parallel simulation.

8.4. THE MINIMUM NODE COVER PROBLEM

Consider the undirected graph $H = (Y, Z)$ of node set Y and edge set Z. The Minimum Node Cover Problem (MNCP) asks for a subset of Y such that every edge in Z has at least one end node in this subset (i.e., the subset is a *node cover*), and furthermore no smaller subset of Y exists satisfying this property (Figure 8.3). MNCP is *NP*-hard, and as such can benefit greatly from heuristic approaches to solve it, as an efficient algorithm to solve it exactly is not expected to exist.

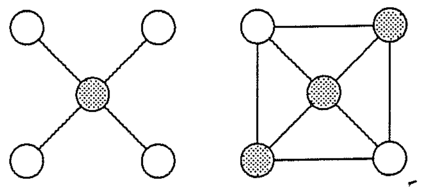


Figure 8.3. *In each of the two graphs, shaded nodes are the members of a minimum node cover.*

A heuristic for MNCP that has become very popular for its interesting theoretical properties is the following. First a maximal matching is found in Z. By definition, every edge in Z has at least one of its end nodes as end node of some edge in the maximal matching. The end nodes of the edges in the maximal matching are then a node cover, and can be taken as an approximation to MNCP. Finding this maximal matching is usually approached by scanning the edges of Z and including an edge in the matching if it does not share an end node with any of the edges already in the matching. The number of edges in the resulting maximal matching can sometimes be reduced if this scan of the edges is done in nonincreasing order of the sum of the degrees of the end nodes of the edges. Similarly to the Christofides heuristic presented in Section 6.3 for TSP, this heuristic for MNCP also has an interesting performance bound. The node cover it produces is never more than twice as large as the minimum node cover.

MNCP can be formulated as an unconstrained optimization problem as follows. Suppose $|Y| = n$, and let us regard the node set Y as a set of 0, 1-variables $y_1, \ldots, y_n$. For notational convenience, we shall make a distinction between a variable y_i and its value d_i in what follows. Consider now the problem of minimizing the function

$$\phi(y) = \sum_{y_i \in Y} y_i + \sum_{(y_i,y_j) \in Z} 2(1-y_i)(1-y_j) \tag{8.12}$$

over $\{0,1\}^n$.

Theorem 8.2. *A subset $W \subseteq Y$ solves MNCP if and only if there exists a global minimum $d^* \in \{0,1\}^n$ of ϕ (of components $d_1^*, \ldots, d_n^*$) such that $d_i^* = 1$ for all $y_i \in W$ and $d_i^* = 0$ for all $y_i \notin W$.*

Proof: Assume first that W solves MNCP, and suppose, to the contrary of the assertion, that setting $d_i = 1$ if and only if $y_i \in W$ does not yield a global minimum of ϕ over $\{0,1\}^n$. In this case, the value of ϕ can be further decreased, which by (8.12) and the optimality of W can only be achieved by setting to 0 some variable with value 1 and whose corresponding node's neighbors in H correspond to variables with value 1. Clearly, if this can be done then W must not solve MNCP.

In order to show the converse statement, assume that $d_i^* = 1$ if and only if $y_i \in W$, and suppose, to the contrary of the assertion, that W does not solve MNCP. Then it must be that either W is not a node cover in H or it is a node cover that is not minimum. In the former case, by (8.12) the value of ϕ can be decreased from what it is at d^* by simply setting to 1 a variable corresponding to an end node of an edge whose end nodes are both outside W. In the latter case, ϕ can be decreased by setting to 1 those variables (and only them) that correspond to nodes in a smaller node cover. In either case, d^* must not be a global minimum of ϕ over $\{0,1\}^n$. ■

In order to solve MNCP, we may, by Theorem 8.2, concentrate solely on minimizing the function ϕ of (8.12) over $\{0,1\}^n$. Interestingly, (8.12) bears some resemblance to (8.2), which gives the energy of a binary Hopfield neural network. Specifically, if we let neuron n_i correspond to variable y_i, and furthermore set $\theta_i = 0$ for

all $n_i \in N$, then E and $\phi(y)$ are exactly the same (except for an additive constant equal to $-n$) if

$$w_{ij} v_i v_j = \begin{cases} -2(1-y_i)(1-y_j), & \text{if } (y_i, y_j) \in Z; \\ 0, & \text{otherwise} \end{cases} \tag{8.13}$$

for all $n_i, n_j \in N$ and

$$e_i v_i = 1 - y_i \tag{8.14}$$

for all $n_i \in N$.

(8.13) and (8.14) suggest a binary Hopfield neural network in which the state of n_i is $1 - y_i$ for all $n_i \in N$. For this network, (8.1) becomes

$$1 - y_i := \text{step}\left(1 - 2 \sum_{n_j | (y_i,y_j) \in Z} (1 - y_j)\right),$$

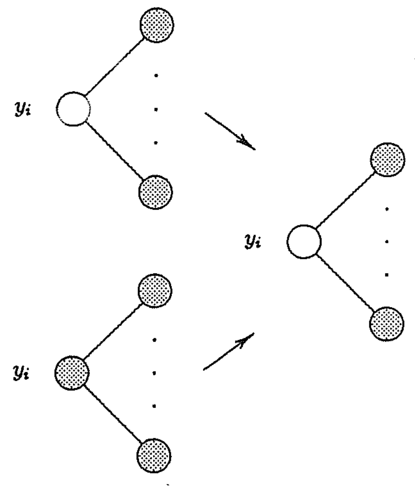
yielding

$$y_i := 1 - \text{step}(1 - 2z_i) \tag{8.15}$$

for all $n_i \in N$, where z_i is the number of neurons n_j such that $(y_i, y_j) \in Z$ and $y_j = 0$. By (8.15), y_i is assigned value 0 if $z_i = 0$ and value 1 if $z_i \geq 1$. In the context of MNCP, this can be regarded as the "local" heuristic (Figure 8.4): if every edge having y_i as an end node is covered ($z_i = 0$), then y_i leaves the node cover ($y_i := 0$); otherwise, i.e., at least one edge having y_i as an end node is not covered ($z_i \geq 1$), then y_i enters the node cover ($y_i := 1$). This guarantees that the variables with value 1 when the network stabilizes are in fact a node cover, although optimality cannot be assured, because, by Theorem 8.1, only local minima of ϕ are guaranteed to be reached. MNCP will also be used as an illustration in Chapter 9, when we consider alternatives to escape from such local minima in the search for a global minimum.

(8.15) can be used directly in the simulation algorithms given in Section 8.2, by replacing occurrences of a neuron n_i's state v_i with y_i, to solve MNCP. A related problem, the maximum independent set problem, asks for a subset of Y having maximum cardinality among the subsets of Y with no nodes connected by an edge. This problem can also be solved concomitantly with MNCP by considering the values of $1 - y_i$ for all $n_i \in N$ when the network stabilizes. As we know, the complement with respect to Y of a node cover is an independent set, so the complement with respect to Y of a minimum node cover is a maximum independent set.

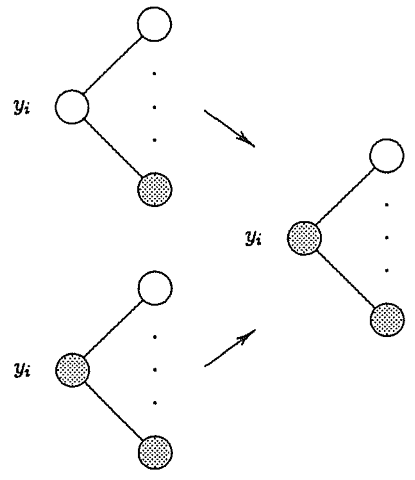
It should be noted that the graph G representing the resulting PC automaton network has exactly the same structure as H. This is so because there is a one-to-one correspondence between nodes in N and variables in Y, and between edges in E and edges in Z, as an edge $(n_i, n_j) \in E$ exists if and only if $w_{ij} \neq 0$, which in the network to solve MNCP holds if and only if $(y_i, y_j) \in Z$ (cf. (8.13)). The fact that the two graphs may be thought of as being the same is important from the standpoint of simulating the PC automaton network by the distributed parallel algorithm we discussed in Section 8.2.2, inasmuch as the level of concurrency that



(a)

Figure 8.4. *If every edge incident to y_i is already covered, as indicated by the shaded nodes, then y_i will not belong to the node cover when it is next updated, regardless of its current status (a). If, on the other hand, at least one of its incident edges is not covered, then y_i will belong to the node cover when it is next updated, also regardless of its current situation (b).*

this algorithm can achieve depends intimately on the structure of G (cf. Section 4.2.3). The structure of G is the backbone architecture for communication during the simulation, and on this architecture processors communicate with neighbors only. For a generic graph problem on H with $Y = N$, as long as neighbors in G do not correspond to nodes in H that can be arbitrarily far apart from each other, there is reason to believe that the distributed parallel simulation of the PC automaton network can yield a good level of concurrency. In the case of MNCP, for example, neighbors in G are neighbors in H, and the distributed parallel simulation will only perform poorly in terms of concurrency if H is already too dense. In general, the goal is that neighbors in G correspond to nodes in H separated by a fixed distance (i.e., the same for all instances of H), equal to one in the case of MNCP. This fixed distance depends of course on the problem's objective function, just as in the case of MNCP it depends on the function ϕ of (8.12). Some problems are described in the literature for which no such fixed distance exists, and others for which the fixed distance is larger than one.



(b)

Figure 8.4 (continued)

8.5. BIBLIOGRAPHIC NOTES

Our treatment of binary Hopfield neural networks in Section 8.1 is based on Hopfield (1982), where these networks were introduced. The distributed parallel algorithm of Section 8.2 is based on Barbosa and Lima (1990).

The issue of associative memories in the context of neural networks is treated in depth by Kohonen (1988). Our treatment of the subject in Section 8.3 follows Hertz, Krogh, and Palmer (1991).

General material on MNCP, including its intractability, can be found in Karp (1972) and Garey and Johnson (1979) (the latter also contains a discussion on the heuristic that yields node covers at most twice as large as the minimum node cover). The formulation given in Section 8.4 for this problem is from Barbosa and Gafni (1989a), and can also be found in Barbosa and Boeres (1990). This formulation is a particular case of a more general formulation that encompasses other difficult combinatorial optimization problems as well (see Barbosa and Gafni (1989a) for this more general formulation, and for a more detailed discussion on how it affects the structure of G — and therefore the amount of concurrency during the PC automaton network simulation).

9

Markov random fields

Markov random fields are the subject of this chapter. They are introduced in Section 9.1, where their equivalence to Gibbs random fields and other important properties are discussed. Simulation algorithms are given in Section 9.2, after a characterization of Markov random fields as PC automaton networks. Additional properties, related to the simulated annealing method, and the corresponding simulation algorithms, are given respectively in Sections 9.3 and 9.4. Other properties are discussed in Section 9.5, and then in Section 9.6 Boltzmann machines, constituting a particular case of Markov random fields, are introduced, along with an application to MNCP (of combinatorial optimization). Bibliographic notes are given in Section 9.7.

9.1. THE MODEL AND BASIC PROPERTIES

In this chapter (and in Chapter 10), there is a random variable v_i associated with each node $n_i \in N$. Each of these variables take values, in this chapter, from a common finite domain D. As in Section 8.4, for notational ease we make a distinction between a variable v_i and its value d_i. Also for notational convenience, unlike our practice in some previous chapters, in this chapter we denote the members of D^n by explicitly listing their components, as in $(d_1, \ldots, d_n)$.

One common element in the discussion in Chapters 9 and 10 will be the concepts of a Markov Random Field (MRF), and of a Gibbs Random Field (GRF). Let V denote the set of variables $\{v_1, \ldots, v_n\}$, and for each $v_i \in V$ define a set of neighbors $Q(v_i)$ in such a way that a *homogeneous neighborhood* is obtained. A homogeneous neighborhood is such that, for any two $v_i, v_j \in V$, if $v_j \in Q(v_i)$, then $v_i \in Q(v_j)$. Let P be a probability distribution on D^n. We say that V is an MRF with respect to Q and P if

$$P(d_1, \ldots, d_n) > 0 \tag{9.1}$$

for all $(d_1, \ldots, d_n) \in D^n$, and

$$P(d_i \mid d_j;\ v_j \neq v_i) = P\big(d_i \bigm| d_j;\ v_j \in Q(v_i)\big) \tag{9.2}$$

for all $v_i \in V$. In (9.2), and henceforth in this chapter and in Chapter 10, the notation $d_j;\ X(v_j)$ stands for the joint occurrence in a point in D^n of d_j for all $v_j \in V$ such that the predicate $X(v_j)$ is true. So $P(d_i \mid d_j;\ v_j \in \mathcal{Q}(v_i))$ is the probability of d_i according to P, conditioned upon the occurrence of d_j for all $v_j \in \mathcal{Q}(v_i)$.

(9.1) and (9.2) represent, respectively, a positivity requirement on the distribution P and a possible generalization of the Markovian dependency beyond the usual single-dimension context. It is possible to do without the positivity condition expressed in (9.1) for most purposes, but we shall not dwell on that issue in this book. MRF's have a great importance in modeling processes where the "local" dependency among variables is a crucial element, as for example within the field of image analysis. In this chapter, however, we shall focus our attention on the consequences of the definition of an MRF within applications to optimization.

Now consider a subset C of the set of random variables V such that, for every pair $v_i, v_j \in C$, $v_j \in \mathcal{Q}(v_i)$ (and conversely, by the definition of $\mathcal{Q}$). Let $\mathbf{C}$ denote the set of all such subsets, and define the *energy* function

$$E(d_1, \ldots, d_n) = \sum_{C \in \mathbf{C}} V_C(d_1, \ldots, d_n) \tag{9.3}$$

for all $(d_1, \ldots, d_n) \in D^n$. In (9.3), V_C is called a *potential* and depends at most on those coordinates in $(d_1, \ldots, d_n)$ that correspond to variables in C. In addition, V_C must not comprise any additive term that depends on a strict subset of C (terms with this characteristic are themselves constituents of potentials), so V_C is either constant or depends on all variables in C. We say that V is a GRF with respect to $\mathcal{Q}$ and P if P is the *Boltzmann-Gibbs distribution*, here denoted by π, i.e.,

$$\begin{aligned} P(d_1, \ldots, d_n) &= \pi(d_1, \ldots, d_n) \\ &= \frac{\exp\bigl(-E(d_1, \ldots, d_n)/T\bigr)}{\sum_{(d'_1, \ldots, d'_n) \in D^n} \exp\bigl(-E(d'_1, \ldots, d'_n)/T\bigr)}, \end{aligned} \tag{9.4}$$

for all $(d_1, \ldots, d_n) \in D^n$. In (9.4), T is a "temperature" parameter (by analogy with the terminology used in statistical physics) whose significance will become clear in Section 9.3.

MRF's and GRF's are related to each other by quite a notable property, which we establish next.

Theorem 9.1. *V is an MRF with respect to the neighborhood $\mathcal{Q}$ and the probability distribution P if and only if it is a GRF with respect to the same neighborhood and the same distribution.*

Proof: See Appendix C. ■

Theorem 9.1 states that the "local" dependency that the definition of an MRF entails (cf. (9.2)) is possible if and only if the distribution P is the Boltzmann-Gibbs distribution π of (9.4). By Theorem 9.1, it is then possible to specify an MRF by giving potentials V_C rather than by giving the so-called *local characteristics* (conditional probabilities) required by its original definition. Given these potentials, the local characteristics are then obtained rather easily via

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{\exp\bigl(-E_i(d_1, \ldots, d_n)/T\bigr)}{\sum_{d'_i \in D} \exp\bigl(-E_i(d_1, \ldots, d'_i, \ldots, d_n)/T\bigr)}, \tag{9.5}$$

for all $v_i \in V$. In (9.5), E_i is like the energy E of (9.3), but only includes potentials that may depend on v_i, i.e.,

$$E_i(d_1, \ldots, d_n) = \sum_{C \in \mathbf{C} \mid v_i \in C} V_C(d_1, \ldots, d_n). \tag{9.6}$$

Still in (9.5), $(d_1, \ldots, d'_i, \ldots, d_n)$ agrees with $(d_1, \ldots, d_n)$ at all coordinates except the ith, where it takes the value d'_i. The reader who ventures into the proof of Theorem 9.1 given in Appendix C will note that it is also possible to obtain the potentials in terms of the local characteristics.

Evaluating π on D^n is in general very costly, frequently impossible in practical situations, although it stands as very desirable in most applications of MRF's. Then the most basic problem to be solved in connection with MRF's is the problem of sampling from the distribution π, which is all one can hope for under the constraints of tolerable computational costs. The process whereby this sampling is achieved is called *Gibbs sampling*. In Gibbs sampling, variables are scanned in any order that ensures that each one of them is visited infinitely often, and upon each visit to variable v_i its value is updated to d_i according to the conditional probability in (9.5). Gibbs sampling entails a discrete-time stochastic process that in this chapter we describe as the sequence $V(0), V(1), \ldots$, where, for $k \geq 0$, $V(k) \in D^n$ has as components the values of the variables in V after the kth *updating round* of the Gibbs sampling process. An updating round of Gibbs sampling consists of the updating of one single variable in V, or the concurrent updating of more than one variable in V. Theorem 9.2 given next establishes a sufficient condition on the long-term behavior of Gibbs sampling when more than one variable is updated concurrently in a same updating round.

Theorem 9.2. *Let $\mathcal{K}_1, \mathcal{K}_2, \ldots$ denote the sets of variables selected for concurrent updating in each round of Gibbs sampling. For $k > 0$, suppose that no two variables $v_i, v_j \in \mathcal{K}_k$ are such that $v_j \in \mathcal{Q}(v_i)$, unless V_C is constant for all $C \in \mathbf{C}$ such that $v_i, v_j \in C$. If every variable in V appears in the sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots$ infinitely often, then*

$$\lim_{k \to \infty} \Pr\bigl(V(k) = (d_1, \ldots, d_n) \mid V(0) = (d'_1, \ldots, d'_n)\bigr) = \pi(d_1, \ldots, d_n),$$

for all $(d_1, \ldots, d_n), (d'_1, \ldots, d'_n) \in D^n$.

Proof: Omitted. ■

Theorem 9.2 states that the process of Gibbs sampling started at any member of D^n converges to the distribution π of (9.4) if no two variables that are neighbors are updated concurrently, except when every potential that may depend on the two neighbors is constant with respect to both of them. As in the case of the binary Hopfield neural networks discussed in Chapter 8, a natural question is whether the violation of this constraint can still lead to convergence to π. To see that this is not the case, consider the following example. Let $V = \{v_1, v_2\}$ and $D = \{0, 1\}$. Suppose E has a very high value whenever $v_1 \neq v_2$, and a very low value otherwise. Then π has a value close to $1/2$ whenever $v_1 = v_2$ and close to 0 when $v_1 \neq v_2$, by (9.4). Also, the conditional probabilities used during the Gibbs sampling (cf. (9.5)) will almost always lead the variables to assume equal values. If the process is started, say, at the point $(0, 1)$, and if the two variables are at all steps allowed to be updated concurrently, then a sequence of pairs like $(0,1), (1,0), (0,1), \ldots$ will almost surely occur, which is in contradiction with the distribution π, according to which the pairs $(0, 0)$ and $(1, 1)$ should be the most frequent. Then the condition that in general no two neighboring variables be updated concurrently is also necessary for convergence to π.

9.2. SIMULATION ALGORITHMS

9.2.1. The sequential algorithm

Theorem 9.2 establishes constraints on the concurrency of variable updates during Gibbs sampling, as well as on how frequently each variable has to be updated, similarly to the case of binary Hopfield neural networks (cf. Chapter 8). The process of Gibbs sampling then characterizes an MRF, along with its neighborhood structure, as another instance of a PC automaton network. In this case, G is obtained by letting $\mathcal{N}(n_i)$ be the set of nodes to which the variables in $\mathcal{Q}(v_i)$ correspond, excluding those nodes n_j such that every potential that might depend on v_i and v_j is constant. In G, the sets of nodes corresponding to the sets of variables $\mathcal{K}_1, \mathcal{K}_2, \ldots$ that appear in the statement of Theorem 9.2 are then independent sets, as required by the definition of a PC automaton network. In addition, each set of variables $C \in \mathbf{C}$ such that V_C is nonconstant corresponds to a clique in G (i.e., a completely connected subgraph). The updating function f of the PC automaton network is such that $x_i(s)$ is given by the initial value of v_i for $s = 0$, and for $s > 0$ by the value of v_i after the most recent pulse in which n_i was in the independent set. The initial value of v_i is henceforth denoted by d_i^0. This PC automaton network can be simulated by a sequential algorithm that scans the variables in V in a fixed order and updates them based on the conditional probabilities of (9.5). This algorithm, given next as Algorithm *Seq_MRF*, runs for a pre-specified number K of full scans, i.e., every variable is updated exactly K times.

```
Algorithm Seq_MRF:
    for $i := 1$ to $n$ do
        $v_i := d_i^0$;
    for $k := 1$ to $K$ do
        for $i := 1$ to $n$ do
            Evaluate the conditional probability
```

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{\exp\left(-(1/T)\sum_{C\in\mathbf{C}|v_i\in C} V_C(d_1,\ldots,d_n)\right)}{\sum_{d_i'\in D}\exp\left(-(1/T)\sum_{C\in\mathbf{C}|v_i\in C} V_C(d_1,\ldots,d_i',\ldots,d_n)\right)}$$

```
            for all $d_i \in D$, and choose a value for $v_i$ accordingly.
```

9.2.2. The distributed parallel algorithm

The process of Gibbs sampling on an MRF is only guaranteed to converge to the Gibbs distribution if no two variables are updated concurrently when at least one potential is nonconstant with respect to both of them, and in addition no variable is indefinitely kept from being updated. An MRF, together with its neighborhood structure, can then be viewed as a PC automaton network.

The distributed parallel simulation of this PC automaton network can then be accomplished by means of an algorithm coined by the template given in Section 4.3.3. In this case, processor p_0 does not participate in the termination detection of the simulation, as every variable is updated a fixed number of times, K. The participation of p_0 is then simply related to logging the simulation's progress and to the computation of statistics. In Section 9.5 (and in Chapter 10), we shall encounter situations in which the participation of p_0 is somewhat more elaborate. In order to fill the template for distributed parallel simulations of PC automaton networks, we must specify the procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for every processor p_i such that $n_i \in N$. From Section 4.3.3, we know that MSG_i contains exactly one message from each of p_i's neighbors. These messages are, in the case of an MRF, the values of the variables corresponding to those neighbors, used to update v_i.

The main steps of the three procedures are given next. Just as Algorithm *Seq_MRF*, they utilize (9.5) for updating the variables.

```
INITIALIZE$_i$:
  1. Let $v_i := d_i^0$;
  2. Send $d_i$ to $p_0$;
  3. Send $d_i$ to every downstream neighbor $p_j$ of $p_i$.
UPDATE$_i$($MSG_i$):
  1. Let $d_j \in MSG_i$ be the value of $v_j$ for all $n_j \in \mathcal{N}(n_i)$;
  2. Evaluate the conditional probability
       $P(d_i \mid d_j;\ v_j \neq v_i) = \frac{\exp\left(-(1/T)\sum_{C \in \mathbf{C} | v_i \in C} V_C(d_1, \ldots, d_n)\right)}{\sum_{d_i' \in D} \exp\left(-(1/T)\sum_{C \in \mathbf{C} | v_i \in C} V_C(d_1, \ldots, d_i', \ldots, d_n)\right)}$
     for all $d_i \in D$, and choose a value for $v_i$ accordingly.
NOTIFY$_i$:
  1. Send $d_i$ to $p_0$;
  2. Send $d_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

In these procedures, d_i is a variable local to p_i.

The concurrency in variable updating that this distributed parallel simulation can provide is of course dependent upon the structure of G, as in the situations we encountered in Chapter 8. In the case of an MRF, the structure of G contains cliques induced by the neighborhood $\mathcal{Q}$. If at least one of these cliques is very large, then G is a very dense graph, and little concurrency is expected. As we know from Section 9.2.1, large cliques appear when potentials V_C depending on a large subset of variables $C \in \mathbf{C}$ occur.

9.3. FURTHER PROPERTIES

9.3.1. Simulated annealing

Gibbs sampling is the basic building block of MRF updating, and by Theorem 9.2 leads the probability distribution over D^n to converge to the Boltzmann-Gibbs distribution of (9.4) if the parameter T is kept constant. In most applications, however, what one wishes is to identify the set of points $D_0^n \subseteq D^n$ where the energy E of (9.3) is globally minimum. This is certainly the case of applications to optimization, as we shall see in Section 9.6, similarly to our approaches to optimization with neural networks in Chapters 6 through 8. One possibility to approach the search for global minima in the context of Gibbs sampling is to reduce the value of T continually from a high starting value, as in *simulated annealing*, which we now describe.

Simulated annealing was proposed as a sequential method to attempt the global minimization of functions by allowing occasional "uphill" moves, i.e., moves in which the value of the function increases during the optimization process. The motivation to seek an optimization method with this characteristic comes from the severe

limitations of gradient-based techniques, which almost invariably get stuck at local minima, often far from a global minimum. This is the case, as we recall, of the approaches we described in Chapters 6 and 8 to optimization based on Hopfield neural networks.

The essence of simulated annealing is quite simple. Starting at an initial point in the solution space, a sequence of points is generated by picking a "neighbor" of the current point. This "neighborhood" relation between points depends intimately on the particular problem at hand, but is often related to the smallest change that a point can suffer to yield another. In general, this smallest change is a change in one of the variables of the problem. When a new point is generated, it may be accepted as the next point of the sequence or rejected. It is accepted if at the new point the function is not higher than at the current point. If the function is higher, then its acceptance characterizes one of the "uphill" moves that the method allows occasionally. Such moves occur probabilistically, and are governed by a function of the difference between the two values of the function (the current and the prospective). Along the process, the parameter T is continually decreased.

Let us now be more specific about the functioning of simulated annealing. Consider a function ϕ on D^n, let $y \in D^n$ be the current point and $y' \in D^n$ be a "neighbor" of y that is being considered as the next point in the sequence. If $\phi(y') \leq \phi(y)$, then y' is accepted and taken as the new current point. If, on the other hand, $\phi(y') > \phi(y)$, then y' is accepted with probability

$$\exp\Big(\big(\phi(y) - \phi(y')\big)/T\Big). \tag{9.7}$$

If y' is not accepted, then y continues to be the current point and another of its "neighbors" is selected, possibly y' once again (Figure 9.1). As long as T is kept constant, this procedure is the *Metropolis algorithm*, introduced decades ago for the simulation of a group of atoms at a fixed temperature T. The use of the probability of (9.7) leads the system to evolve according to the Boltzmann-Gibbs distribution of (9.4). We should note that the Metropolis algorithm, when used for simulating physical systems, employs the parameter T as the system's temperature, and then (9.7) is modified by the inclusion of *Boltzmann's constant*, k_B, as in

$$\exp\Big(\big(\phi(y) - \phi(y')\big)/k_B T\Big).$$

Our interest in this book is, however, in the use of this algorithm (and other related ones) on problems of computer science, so T has no meaning as a temperature and Boltzmann's constant can be done away with for notational convenience, as in (9.4) and (9.7). We do, however, sometimes refer to T as a temperature, even outside the proper context of a physical system.

The crux of the simulated annealing method is a simple modification of the Metropolis algorithm, whereby, along the simulation, the parameter T is slowly decreased, in an attempt to imitate the physical *annealing* process. The annealing of a material is a laboratory procedure whereby the material is brought to a state of

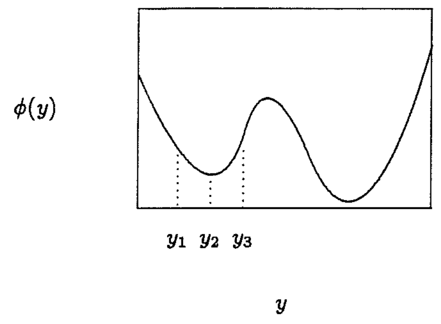


Figure 9.1. *If y_1 is the current point and y_2 is the prospective next point, then y_2 is definitely accepted as the new current point, as $\phi(y_2) < \phi(y_1)$. If, on the other hand, y_2 is the current point and y_3 is being considered as a possible next point, then y_3 is taken as the new current point probabilistically, as $\phi(y_3) > \phi(y_2)$.*

ground energy from an initial state of very high energy. This can only be achieved if the material is cooled very slowly, particularly at low temperatures, lest the system becomes stuck at a state whose energy is not the lowest possible. Simulated annealing employs the same temperature-reduction mechanism in the search for global minima of ϕ during the execution of the Metropolis algorithm. At high values of T (i.e., initially), practically every new point generated is accepted, by (9.7). At low values of T (i.e., near the end of the simulation), practically no increase in the value of the function is any longer accepted, also by (9.7).

Next we provide Algorithm *Seq_Simulated_Annealing*, a pseudo-code for simulated annealing. The parameter T starts off at a value T_0, and is gradually reduced toward its final value, T_f, at which time the algorithm terminates. For each value of T, the algorithm iterates (i.e., generates new points) $L(T)$ times. The value of T is reduced by means of a function $r(T)$, which we leave for now unspecified; it will be discussed in more detail in Section 9.3.2. We take $y^0 \in D^n$ to be the starting point.

Algorithm *Seq_Simulated_Annealing*:

```
y := $y^0$;
T := $T_0$;
while $T \geq T_f$ do
    begin
        for $\ell$ := 1 to $L(T)$ do
            begin
                Generate $y'$;
                if $\phi(y') \leq \phi(y)$ then
                    $y := y'$
                else
                    Assign $y'$ to $y$ with probability
                    $\exp\Big((\phi(y) - \phi(y'))/T\Big)$
            end;
        $T := r(T)$
    end.
```

There exist numerous variations of this basic procedure, embodying several different choices for T_0, T_f, $r(T)$, $L(T)$, and even the stopping criterion.

9.3.2. Gibbs sampling and simulated annealing

Gibbs sampling can be modified quite easily to accommodate a gradual reduction of the parameter T, following our discussion on simulated annealing in Section 9.3.1. For such, it suffices that T obey a *cooling schedule* according to which it decreases continually as the process goes on, in a fashion entirely analogous to the use of the reduction function $r(T)$ of Algorithm *Seq_Simulated_Annealing*. The following result describes sufficient conditions on the reduction of T so that the members of D_0^n, the set of points of D^n at which the energy E of (9.3) is globally minimum, are eventually identified.

Theorem 9.3, like Theorem 9.2, is stated within the context of distributed parallel processing, and can in this sense be regarded as a "variable-temperature" companion of Theorem 9.2, which established the convergence of Gibbs sampling for a fixed value of T.

Theorem 9.3. *Let $\mathcal{K}_1, \mathcal{K}_2, \ldots$ denote the sets of variables selected for concurrent updating in each round of Gibbs sampling, and $T(1), T(2), \ldots$ the corresponding sequence of values of T. For $k > 0$, suppose that no two variables $v_i, v_j \in \mathcal{K}_k$ are such that $v_j \in \mathcal{Q}(v_i)$, unless V_C is constant for all $C \in \mathbf{C}$ such that $v_i, v_j \in C$. If every variable in V appears in the sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots$ infinitely often, and $T(k) \to 0$ as $k \to \infty$, and furthermore $T(k) \geq n\Delta/\ln k$ for all $k > 1_r$ where Δ is the largest possible gap between the values of E of (9.3) at two different points in D^n, then*

$$\lim_{k\to\infty} \Pr\big(V(k) = (d_1, \ldots, d_n) \mid V(0) = (d'_1, \ldots, d'_n)\big) = \pi_0(d_1, \ldots, d_n),$$

for all $(d_1, \ldots, d_n), (d'_1, \ldots, d'_n) \in D^n$, *where* π_0 *is the uniform distribution on* D_0^n.

Proof: Omitted. ■

As in the case of Theorem 9.2, the condition expressed in Theorem 9.3 regarding the concurrent updating of variables is not only sufficient but also necessary. The reason for this is precisely the same as we provided in the case of Theorem 9.2.

This version of Theorem 9.3 is slightly weaker than the one presented in the literature, as it requires a slightly higher lower bound on the cooling schedule $T(1), T(2), \ldots$. In the literature, we find essentially the same statement, except that a raster scan is performed on the set V for updating the variables, so the value of k is there consistently larger than ours. Both versions, however, give us unreasonable cooling schedules for practical purposes, since the number of updates required to reach a final temperature T_f grows exponentially as T_f is made lower. This can be seen rather easily by observing the inequality appearing in the statement of Theorem 9.3, which yields

$$\exp(n\Delta/T_f)$$

as the minimum number of updates before T_f is reached. In practice, one possibility that is largely favored is to let T be decreased by a constant factor whenever a variable is updated. In the context of our simulation algorithm, this might translate, for instance, into the cooling schedule

$$T_k = \alpha T_{k-1}, \tag{9.8}$$

where $0 < \alpha < 1$, T_0 is a high initial value, and $k - 1 > 0$ is the number of times a variable has been updated. Clearly, the cooling schedule in (9.8) allows every variable to be updated at all the temperatures $T_0, T_1, \ldots$, but no more than

$$1 + \left\lfloor \log_{1/\alpha}\left(\frac{T_0}{T_f}\right)\right\rfloor \tag{9.9}$$

times before T_f is reached, which is, we recognize, considerably better than the number of updates implied by Theorem 9.3, although we have of course lost the guarantee of convergence to the global minima.

9.4. FURTHER SIMULATION ALGORITHMS

9.4.1. The sequential algorithm

The cooling schedule given by (9.8) can be used to incorporate simulated annealing in Gibbs sampling. Although this cooling schedule no longer complies with the requirements established by Theorem 9.3 for convergence to the global minima of the energy function, the remaining requirements still characterize the MRF, along with its neighborhood structure, as a PC automaton network, in precisely the same way as we discussed in Section 9.2.1 (except of course for the fact that the updating function f now depends on a varying temperature T). A sequential algorithm to

simulate this PC automaton network employing the approximate cooling schedule is Algorithm *Seq_Annealed_MRF*, given next. Variable updates are done according to (9.5).

```
Algorithm Seq_Annealed_MRF:
    for $i := 1$ to $n$ do
        $v_i := d_i^0$;
    $T := T_0$;
    while $T \geq T_f$ do
        begin
            for $i := 1$ to $n$ do
                Evaluate the conditional probability

                $P(d_i \mid d_j;\ v_j \neq v_i) = \dfrac{\exp\left(-(1/T)\sum_{C \in \mathbf{C} | v_i \in C} V_C(d_1, \ldots, d_n)\right)}{\sum_{d'_i \in D} \exp\left(-(1/T)\sum_{C \in \mathbf{C} | v_i \in C} V_C(d_1, \ldots, d'_i, \ldots, d_n)\right)}$

                for all $d_i \in D$, and choose a value for $v_i$ accordingly;
            $T := \alpha T$
        end.
```

In Algorithm *Seq_Annealed_MRF*, every variable is updated a number of times given as in (9.9).

9.4.2. The distributed parallel algorithm

The distributed parallel counterpart of Algorithm *Seq_Annealed_MRF* of Section 9.4.1 can be obtained rather directly from the distributed parallel algorithm given in Section 9.2.2 for Gibbs sampling. The main steps of the three procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for processor p_i are given next. As in section 9.2.2, each member of MSG_i is the value of a variable corresponding to one of p_i's neighbors, to be used in updating v_i. Processor p_0, as before, does not participate in detecting the termination of the simulation, as every variable is updated a fixed number of times K, in this case given as in (9.9).

```
INITIALIZE_i:
    1. $v_i := d_i^0$;
    2. Send $d_i$ to $p_0$;
    3. Send $d_i$ to every downstream neighbor $p_j$ of $p_i$;
    4. $T := T_0$.
UPDATE_i(MSG_i):
    1. Let $d_j \in MSG_i$ be the value of $v_j$ for all $n_j \in \mathcal{N}(n_i)$;
    2. Evaluate the conditional probability
       $P(d_i \mid d_j;\ v_j \neq v_i) = \frac{\exp\left(-(1/T)\sum_{C\in\mathbf{C}|v_i\in C} V_C(d_1,\ldots,d_n)\right)}{\sum_{d_i'\in D}\exp\left(-(1/T)\sum_{C\in\mathbf{C}|v_i\in C} V_C(d_1,\ldots,d_i',\ldots,d_n)\right)}$
       for all $d_i \in D$, and choose a value for $v_i$ accordingly;
    3. $T := \alpha T$.
NOTIFY_i:
    1. Send $d_i$ to $p_0$;
    2. Send $d_i$ to every $p_j$ such that $n_j \in \mathcal{N}(n_i)$.
```

In these procedures, d_i is a variable local to p_i.

9.5. YET ANOTHER PROPERTY

Another interesting property of the basic Gibbs sampling given in Section 9.1 is characterized in the next theorem. What the theorem states is that, given any function g on D^n, the time average of g along the process of Gibbs sampling converges to the expected value of g according to the Boltzmann-Gibbs distribution π of (9.4). This property is often referred to as the *ergodicity* of Gibbs sampling, by analogy with the ergodicity of stationary processes.

Theorem 9.4. *Let $\mathcal{K}_1, \mathcal{K}_2, \ldots$ denote the sets of variables selected for concurrent updating in each round of Gibbs sampling, and g any function on D^n. For $k > 0$, suppose that no two variables $v_i, v_j \in \mathcal{K}_k$ are such that $v_j \in \mathcal{Q}(v_i)$, unless V_C is constant for all $C \in \mathbf{C}$ such that $v_i, v_j \in C$. If every variable in V appears in the sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots$ infinitely often, then*

$$\lim_{K\to\infty} \frac{1}{K}\sum_{k=1}^{K} g(V(k)) = \sum_{(d_1,\ldots,d_n)\in D^n} g(d_1,\ldots,d_n)\pi(d_1,\ldots,d_n)$$

holds with probability one.

Proof: Omitted. ■

As in the case of Theorems 9.2 and 9.3, and by the same reasons, the condition in Theorem 9.4 that concurrency in variable updating be restricted to variables upon which no potential depends is necessary in addition to being sufficient.

The time average of the function g of Theorem 9.4 can in practice be computed by processor p_0, as this processor receives from the other processors every updated value of all variables. This possibility was alluded to in Section 9.2.2, and will be important in the case of Bayesian networks (see Chapter 10). More specifically, recall that every single variable update can be regarded as a Gibbs sampling round. The role of p_0 when computing the time average of g is then to maintain a (partial) sum of the values of g, updated whenever an updated value for a variable is received. When the K updates per variable expire, p_0 can then compute the time average of g by simply dividing the sum it maintained by nK. Sometimes, as for example in the case to be discussed in Section 10.2.3, g is a function of one of the variables only, and then the partial sum maintained by p_0 has to be updated only when updated values for that variable are received. At the end of the simulation, the division is then by K, as this is the number of values used to yield the final sum.

9.6. BOLTZMANN MACHINES

9.6.1. The model and basic properties

As we saw in Section 9.1, a GRF is characterized by the Boltzmann-Gibbs distribution (cf. (9.4)) when the energy function has the form given as in (9.3) by a sum of potentials V_C for all $C \in \mathbf{C}$. The particular case of a GRF on $D = \{0,1\}$ with potentials

$$V_C(d_1,\ldots,d_n) = \begin{cases} -\varphi_{ij}d_id_j, & \text{if } C = \{v_i,v_j\} \text{ for some } v_i,v_j \in V; \\ -\psi_i d_i, & \text{if } C = \{v_i\} \text{ for some } v_i \in V; \\ 0, & \text{otherwise} \end{cases} \tag{9.10}$$

for all $C \in \mathbf{C}$ is known as a *Boltzmann machine*. The energy function of a Boltzmann machine is, by (9.3) and (9.10), given by

$$E(d_1,\ldots,d_n) = -\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \varphi_{ij}d_id_j - \sum_{i=1}^{n}\psi_i d_i, \tag{9.11}$$

which is easily recognized as the energy function of a binary Hopfield neural network (cf. (8.2)) with $w_{ij} = \varphi_{ij}$ and $w_{ii} + e_i - \theta_i = \psi_i$. So the application to a Boltzmann machine of the decreasing-temperature Gibbs sampling procedure discussed in Section 9.4 is, by Theorem 9.3, a means to overcome the characteristic of a binary Hopfield neural network that only local minima of the energy are reached, when a global minimum is sought, as implied by Theorem 8.1.

For the general case of a GRF on $\{0,1\}^n$, let $\Delta_i E(d_1,\ldots,d_n)$ denote the difference from the value of the energy when $d_i = 0$ to the value when $d_i = 1$ if all other coordinates are kept constant. We then have, by (9.5), that

$$\begin{aligned}P(d_i = 1 \mid d_j;\ v_j \neq v_i) &\\ = \frac{\exp(-E_i(d_1,\ldots,d_i=1,\ldots,d_n)/T)}{\exp(-E_i(d_1,\ldots,d_i=0,\ldots,d_n)/T)+\exp(-E_i(d_1,\ldots,d_i=1,\ldots,d_n)/T)} &\\ = \frac{\exp(-\Delta_i E(d_1,\ldots,d_n)/T)}{1+\exp(-\Delta_i E(d_1,\ldots,d_n)/T)} & \qquad (9.12)\end{aligned}$$

and

$$\begin{aligned}P(d_i = 0 \mid d_j;\ v_j \neq v_i) &\\ = \frac{\exp(-E_i(d_1,\ldots,d_i=0,\ldots,d_n)/T)}{\exp(-E_i(d_1,\ldots,d_i=0,\ldots,d_n)/T)+\exp(-E_i(d_1,\ldots,d_i=1,\ldots,d_n)/T)} &\\ = \frac{1}{1+\exp(-\Delta_i E(d_1,\ldots,d_n)/T)} & \qquad (9.13)\end{aligned}$$

for all $v_i \in V$, where E_i is given as in (9.6).

For the Boltzmann machine, in particular, (9.11) yields

$$\Delta_i E(d_1,\ldots,d_n) = -\sum_{j\neq i}\varphi_{ij}d_j - \psi_i, \qquad (9.14)$$

so that (9.12) and (9.13) become, respectively,

$$P(d_i = 1 \mid d_j;\ v_j \neq v_i) = \frac{\exp\left(\left(\sum_{j\neq i}\varphi_{ij}d_j + \psi_i\right)/T\right)}{1+\exp\left(\left(\sum_{j\neq i}\varphi_{ij}d_j + \psi_i\right)/T\right)} \qquad (9.15)$$

and

$$P(d_i = 0 \mid d_j;\ v_j \neq v_i) = \frac{1}{1+\exp\left(\left(\sum_{j\neq i}\varphi_{ij}d_j + \psi_i\right)/T\right)} \qquad (9.16)$$

for all $v_i \in V$. Now consider a binary Hopfield neural network with $w_{ij} = \varphi_{ij}$ and $w_{ii} + e_i - \theta_i = \psi_i$. By (8.1) and (9.14), in this neural network the value of v_i would be given by

$$v_i := \mathrm{step}(-\Delta_i E(d_1,\ldots,d_n)), \qquad (9.17)$$

i.e., it would equal 0 if $\Delta_i E(d_1,\ldots,d_n) \geq 0$, 1 if $\Delta_i E(d_1,\ldots,d_n) < 0$. In accordance with our earlier anticipation in this section, such is also the behavior implied by (9.15) and (9.16) for very low values of T. When T has a high value, though, (9.15) and (9.16) force the system to "disobey" the binary Hopfield rule of (8.1) with probability approximately equal to 1/2, hence the desired hill-climbing behavior.

9.6.2. The minimum node cover problem

Boltzmann machines can be used in a variety of problems, and when used within artificial intelligence can make use of a learning algorithm that tries to adjust probability distributions. In this chapter, however, our interest is to employ Boltzmann machines to the solution of optimization problems, based on our discussion in Section 9.6.1 that a Boltzmann machine can in fact be regarded as a combination of a binary Hopfield neural network with simulated annealing.

In order to illustrate the use of Boltzmann machines in optimization, we draw on the same example used in Chapter 8, namely MNCP. This problem was introduced in Section 8.4, where the application of a binary Hopfield neural network to solve it was discussed. If now we employ the variables $v_1,\ldots,v_n$ in place of the variables $y_1,\ldots,y_n$ used in that section, then the action of a binary Hopfield neural network on MNCP, given by (8.15), can be summarized by

$$v_i := 1 - \mathrm{step}(1-2z_i),$$

or, equivalently,

$$v_i := \mathrm{step}(-(1-2z_i)), \qquad (9.18)$$

for all $v_i \in V$. (9.14), (9.17), and (9.18) yield

$$\sum_{j\neq i}\varphi_{ij}d_j + \psi_i = -(1-2z_i),$$

and then (9.15) and (9.16) become, respectively,

$$P(d_i = 1 \mid d_j;\ v_j \neq v_i) = \frac{\exp(-(1-2z_i)/T)}{1+\exp(-(1-2z_i)/T)} \qquad (9.19)$$

and

$$P(d_i = 0 \mid d_j;\ v_j \neq v_i) = \frac{1}{1+\exp(-(1-2z_i)/T)} \qquad (9.20)$$

for all $v_i \in V$.

The conditional probabilities in (9.19) and (9.20) are then the ones to be used in the algorithms of Section 9.4 in order to solve MNCP in a way that does not lead to local minima only. It may be instructive to check that the behavior implied by (9.19) and (9.20) at very low values of T is, in agreement with our discussion at the end of Section 9.6.1, consonant with the behavior of a binary Hopfield neural network on MNCP, discussed in Section 8.4 (Figure 8.4).

9.7. BIBLIOGRAPHIC NOTES

MRF's and GRF's are treated in detail by Spitzer (1971), Grimmett (1973), Besag (1974), Griffeath (1976), Kinderman and Snell (1980), and Isham (1981). The extension to the case in which the positive requirement on the distribution is eliminated is found in Moussouris (1974). The process of Gibbs sampling on an MRF discussed in Section 9.1 is from Geman and Geman (1984).

The algorithm for distributed parallel simulation of Section 9.2 is from Barbosa and Gafni (1989a), where it is presented for the updating of MRF's in the context of combinatorial optimization.

The Metropolis algorithm presented in Section 9.3 was first described by Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953). Simulated annealing was introduced by Kirkpatrick, Gelatt, and Vecchi (1983), and analyzed for convergence and short-term properties by Mitra, Romeo, and Sangiovanni-Vincentelli (1986). Early experimental evaluations of the method include Aragon, Johnson, McGeoch, and Schevon (1984) (which later appeared with considerably more detail in Johnson, Aragon, McGeoch, and Schevon (1989, 1991), with a third part to appear) and Felten, Karlin, and Otto (1985), the latter having been one of the first attempts to implement the method on a parallel machine, aiming at a solution to TSP. This was an approximate approach, in the sense that it appears to have allowed neighbor variables to be updated concurrently. Approximate are also the more recent proposals to be found in Greening (1990, 1991), and for the same reason. A comprehensive annotated bibliography on simulated annealing is given by Collins, Eglese, and Golden (1988). The extension of Gibbs sampling to accommodate the varying-temperature behavior of simulated annealing is from Geman and Geman (1984). The approximate cooling schedule suggested in Section 9.3 was already proposed by Kirkpatrick, Gelatt, and Vecchi (1983), and seems to be used widely (Collins, Eglese, and Golden, 1988).

The algorithm described in Section 9.4 for the distributed parallel simulation of the PC automaton network corresponding to the updating of an MRF under varying temperature is from Barbosa and Gafni (1989a).

The material on ergodicity given in Section 9.5 is based on the ergodicity of stationary processes (see, for example, Karlin and Taylor (1975)), and follows Geman and Geman (1984), where the omitted proofs of Theorems 9.2 through 9.4 can also be found. Their proofs do not, however, directly apply to the case of multiple concurrent variable updates, although an extension in this direction is straightforward.

Boltzmann machines were introduced in Hinton, Sejnowski, and Ackley (1984), and can also be found in Ackley, Hinton, and Sejnowski (1985), where the emphasis is on a learning procedure.

The approach to MNCP of Section 9.6 was described by Barbosa and Gafni (1989a) and experimentally evaluated by Barbosa and Boeres (1990). One interesting aspect discussed in Barbosa and Gafni (1989a) (see also the earlier account by Gafni and Barbosa (1986)) is that an additional computation, to be carried out after termination of the simulated annealing process (or concurrently with it, as long as a few precedences are respected), should be conducted to identify the global state at which the overall minimum of the objective function found during the simulation occurred. These supplementary calculations are based on a maximum-flow computation on the precedence graph associated with the edge-reversal process, but

for MNCP have been found to be unnecessary, as almost invariably the global state at which the simulation stops is also the one at which the overall minimum found during the simulation occurs (Barbosa and Boeres, 1990). This may not be so for other problems, in which case the maximum-flow computation may be needed. References on the traditional approaches to finding the maximum flow on a network are Ford and Fulkerson (1962), Lawler (1976), Even (1979), and Papadimitriou and Steiglitz (1982). More recent approaches, including some that can be naturally implemented in a distributed parallel setting, are described in the surveys by Ahuja, Magnanti, and Orlin (1989) and Goldberg, Tardos, and Tarjan (1990), and in Cormen, Leiserson, and Rivest (1990). An experimental evaluation of some of the distributed parallel approaches can be found in Portella and Barbosa (1992), where the algorithm by Awerbuch (1985b) and two variations of the algorithm by Goldberg and Tarjan (1988) are investigated (an earlier reference on the latter algorithm is Goldberg and Tarjan (1986)).

10

Bayesian networks

Bayesian networks are discussed in this chapter, beginning with their introduction, along with their basic properties, in Section 10.1. One of these properties relates Bayesian networks to fixed-temperature Gibbs random fields, thereby characterizing them as PC automaton networks as well. Simulation algorithms are then given in Section 10.2. Additional properties, this time relating Bayesian networks to Gibbs random fields with varying temperature, are discussed in Section 10.3, and the corresponding simulation algorithms are given in Section 10.4. An application of Bayesian networks to the resolution of lexical ambiguities in the treatment of natural languages is presented in Section 10.5. Bibliographic notes follow in Section 10.6.

10.1. THE MODEL AND BASIC PROPERTIES

In this chapter, we continue with most of the notational conventions adopted in Chapter 9. Specifically, there exists a random variable v_i associated with each node $n_i \in N$, and we make a distinction between a variable v_i and its value d_i. The set of the n random variables is denoted by V. Variables take values from a common finite domain D, in this chapter assumed to be $D = \{0, 1\}$. This is not a necessary requirement, but allows some convenient notational simplifications, while still allowing most interesting applications of Bayesian network models to be addressed. Members of $\{0,1\}^n$ are often denoted by explicitly listing their components, as in $(d_1, \ldots, d_n)$, unlike our practice throughout most of the book (except in Chapter 9). The use of probabilities, as in Chapter 9, continues to dominate the scene, and the notation $d_j;\ X(v_j)$ continues to denote the joint occurrence in a point in $\{0,1\}^n$ of d_j for all $v_j \in V$ such that the predicate $X(v_j)$ is true.

A *Bayesian network* is an acyclic directed graph whose node set is the set of random variables V, and whose directed edges represent direct causal relationships among variables. As nodes in an acyclic directed graph, the variables of a Bayesian network have specific "roles" with respect to one another, and the resulting relationships between them lie at the bottom of many of the results about the model's properties. For $v_i \in V$, let $\mathcal{P}(v_i)$ denote the set of v_i's *parents* in the directed graph (defined to be the variables v_j such that a directed edge from v_j to v_i exists), $\mathcal{C}(v_i)$ the set of v_i's *children* (defined to be the variables v_j such that a directed edge from v_i to v_j exists), and $\mathcal{M}(v_i)$ the set of v_i's *mates* (defined to be the variables v_j such that a directed edge exists from v_i to v_k and from v_j to v_k for some v_k). It is important for us to note right away that the directions of a Bayesian network's edges have nothing to do with the acyclic orientations of G introduced in Chapter 4. Later in this chapter, the two types of edge orientation will be used together, but no confusion should arise, as they stand for entirely different concepts.

Bayesian networks constitute an important tool to model uncertain evidential reasoning in artificial intelligence, i.e., reasoning based on evidences about a domain that relate to each other imprecisely. In order to construct a Bayesian network model for some domain, variables judged to have a direct causal influence on other variables are connected directly to those variables by means of forward directed edges (i.e., they become those variables' parents). The "strengths" of these causal influences are coded, for each variable, by the set of probabilities that the variable has value 0 or 1, conditioned upon the various combinations of values of its parents.

Associated with each $v_i \in V$, there are then the $2^{1+|\mathcal{P}(v_i)|}$ positive conditional probabilities

$$P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \tag{10.1}$$

for all $d_i \in \{0,1\}$. In strict terms, the positivity requirement is in general inessential. But it does allow most notions to be expressed more clearly than otherwise, thence the assumption. For those variables v_i such that $\mathcal{P}(v_i) = \emptyset$ (at least one of these has got to exist, as the directed graph is acyclic), the two conditional probabilities associated with v_i are in fact "unconditional," and are called v_i's *prior probabilities*. The conditional probabilities associated with v_i encode some sort of "expert knowledge" about the domain being modeled. The fact that the Bayesian network is constructed in such a way that every direct causal relationship perceived by its designer is represented by directed edges between pairs of variables means that these conditional probabilities summarize the various (possibly indirect) causal influences among variables.

Let us for a moment assume, for notational ease, that, for all $v_i, v_j \in V$, $v_j \in \mathcal{P}(v_i)$ if and only if $j < i$. In other words, we assume that the variables are totally ordered in a way that complies with the order implied by the directed paths in the network. Under this ordering, the choice made by the network's designer concerning the parent sets bears the intuitive expectation that every variable be independent, given the values of its parents, of all the other variables that precede it (these are variables that cannot be reached from it in the graph, its "nondescendants"). This intuition is confirmed by a more formal approach to the study of Bayesian networks in which they are defined in terms of how the structure of the directed graph and the probabilistic independencies among the variables according to the distribution P relate to each other. This approach is described in the literature, and indeed gives rise to an expression of our intuition as

$$P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) = P(d_i \mid d_j;\ j < i) \tag{10.2}$$

for all $v_i \in V$. So although the distribution P is generally unknown over the entirety of $\{0,1\}^n$ to begin with, by using (10.2) it is possible to show that P is uniquely determined by the conditional probabilities in (10.1). For such, we first write P as

$$\begin{aligned} P(d_1,\ldots,d_n) &= P(d_1)\frac{P(d_1,d_2)}{P(d_1)}\cdots\frac{P(d_1,\ldots,d_n)}{P(d_1,\ldots,d_{n-1})} \\ &= P(d_1)P(d_2 \mid d_1)\cdots P(d_n \mid d_1,\ldots,d_{n-1}), \end{aligned}$$

and then by (10.2) we obtain

$$P(d_1,\ldots,d_n) = \prod_{i=1}^{n} P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \tag{10.3}$$

for all $(d_1,\ldots,d_n) \in \{0,1\}^n$.

Some of the variables in V are special, in the sense that their values are obtained from observations of the domain, and can therefore be regarded as being fixed as far as the process of solving the model is concerned. These variables constitute the *input*, or the *evidences*, to the network. The set of evidences is denoted by $\mathcal{E}$. The fundamental problem to be solved in connection with a Bayesian network is to evaluate the probabilities of each variable $v_i \in V - \mathcal{E}$, conditioned upon the values of the variables in $\mathcal{E}$, that is,

$$P(d_i \mid d_j;\ v_j \in \mathcal{E}), \tag{10.4}$$

for all $v_i \in V - \mathcal{E}$ (these probabilities are called *posterior probabilities*). Once this problem is solved, the network can then be employed in a variety of situations in which diagnostics are sought, albeit imprecise, given a set of evidences (or symptoms, as they ought to be called in some fields). The distribution in (10.3) implies the following theorem.

Theorem 10.1. *The joint distribution over the variables in $V - \mathcal{E}$, conditioned upon the values of the variables in $\mathcal{E}$, is*

$$P(d_i;\ v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

where α is a normalizing constant, for all $(d_i;\ v_i \in V - \mathcal{E}) \in \{0,1\}^{|V-\mathcal{E}|}$.

Proof: We have

$$P(d_i;\ v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \frac{P(d_1,\ldots,d_n)}{P(d_j;\ v_j \in \mathcal{E})},$$

which by (10.3) becomes

$$P(d_i;\ v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))$$

with

$$\alpha = \frac{1}{P(d_j;\ v_j \in \mathcal{E})}.$$

That α can be regarded as a normalizing constant comes from the fact that it does not depend on d_i for any $v_i \in V - \mathcal{E}$. ■

The distribution of Theorem 10.1 is known as the *joint posterior distribution*, given $\mathcal{E}$.

Once the joint distribution over the variables in $V - \mathcal{E}$, conditioned upon the values of the variables in $\mathcal{E}$, is known, it is then possible to evaluate the conditional probabilities in (10.4) quite simply, as in

$$\begin{aligned} P(d_i \mid d_j;\ v_j \in \mathcal{E}) &= \sum_{\substack{(d_\ell;\ v_\ell \in V-\mathcal{E}-\{v_i\}) \\ \in\{0,1\}^{|V-\mathcal{E}|-1}}} P(d_j;\ v_j \in V - \mathcal{E} \mid d_k;\ v_k \in \mathcal{E}) \\ &= \alpha \sum_{\substack{(d_\ell;\ v_\ell \in V-\mathcal{E}-\{v_i\}) \\ \in\{0,1\}^{|V-\mathcal{E}|-1}}} \prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)), \qquad (10.5) \end{aligned}$$

for all $v_i \in V$, by Theorem 10.1. Note that in (10.5) only the originally available conditional probabilities are employed, i.e., those in (10.1).

It is unfortunate, however, that, unless the network has a very simple structure (a tree with all edges directed from a root toward the leaves, so that each variable except the root has exactly one parent), evaluating the probabilities in (10.4) by applying (10.5) is a computationally intractable (i.e., *NP*-hard) problem. This has led to the introduction of inexact methods of solution whereby the probabilities in (10.4) can be assessed on an approximate basis. One of these methods is a *stochastic simulation* technique, which is not the most efficient one in many situations, but has in the context of this book the appeal of relating intimately to the material of Chapter 9. There are essentially two variations of the stochastic simulation of a Bayesian network. They are described in Section 10.2.3, but both rely on a mechanism for updating the variables according to conditional probabilities, given the values of all other variables. Theorem 10.2 given next shows that the variables that matter when updating a variable v_i are those in the neighborhood

$$\mathcal{L}(v_i) = \mathcal{P}(v_i) \cup \mathcal{C}(v_i) \cup \mathcal{M}(v_i). \qquad (10.6)$$

Theorem 10.2. *Let v_i be a variable in V. Then*

$$P(d_i \mid d_j;\ v_j \neq v_i) = \alpha P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

where α is a normalizing constant.

Proof: We have

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{P(d_1, \ldots, d_n)}{P(d_j;\ v_j \neq v_i)},$$

which by (10.3) becomes

$$\begin{aligned} P(d_i \mid d_j;\ v_j \neq v_i) &= \frac{\prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))}{P(d_j;\ v_j \neq v_i)} \\ &= P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)) \\ &\qquad \frac{\prod_{v_j \in V-\{v_i\}-\mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))}{P(d_j;\ v_j \neq v_i)} \\ &= \alpha P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)), \end{aligned}$$

where

$$\alpha = \frac{\prod_{v_j \in V-\{v_i\}-\mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))}{P(d_j;\ v_j \neq v_i)}.$$

That α can be regarded as a normalizing constant comes from the fact that it does not depend on d_i. ■

Note, in the statement of Theorem 10.2, that the members of $\mathcal{P}(v_j)$ inside the product are also members of $\mathcal{M}(v_i)$, hence the dependency of the conditional probability on the entire neighborhood $\mathcal{L}(v_i)$ of (10.6).

The equation appearing in the statement of Theorem 10.2 can be rewritten with the purpose of making the constant α explicit. If $v_i \in V$, then for $v_j \in \mathcal{C}(v_i)$ let

$$\rho_j^0 = P(d_j \mid d_i = 0, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j)) \qquad (10.7)$$

and

$$\rho_j^1 = P(d_j \mid d_i = 1, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j)). \qquad (10.8)$$

Then, using (10.7) and (10.8), the probability that v_i has value d_i, conditioned upon the values of all other variables in V, is, by Theorem 10.2, given by

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i}}{\sum_{d \in \{0,1\}} P(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d}} \qquad (10.9)$$

(some readers may wish to check the reciprocal of the denominator of (10.9) against the α inside the proof of Theorem 10.2 to see that they are the same). The basic mechanism of stochastic simulation is then the following. Scan the variables in $V - \mathcal{E}$ in any order that ensures that every variable is visited infinitely often, and upon each visit to variable v_i update it to d_i according to the conditional probability in (10.9).

We shall henceforth utilize some of the nomenclature introduced in Chapter 9; the reader is then referred to that chapter, especially to Section 9.1, for the appropriate definitions and discussion. The following theorem states that stochastic simulation is a special case of the Gibbs sampling process discussed in Chapter 9 in which we let $\mathcal{Q}(v_i) = \mathcal{L}(v_i)$ for all $v_i \in V$. Recall from Chapter 9 that $\mathcal{Q}$ is a neighborhood structure on V and that $\mathbf{C}$ stands for the set of all subsets C of V whose members are all neighbors of one another by $\mathcal{Q}$. For $C \in \mathbf{C}$, consider the potentials

$$V_C(d_1, \ldots, d_n) = \begin{cases} -\ln P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)), & \text{if } C = \{v_i\} \cup \mathcal{P}(v_i) \\ & \text{for some } v_i \in V; \\ 0, & \text{otherwise.} \end{cases} \qquad (10.10)$$

Note that all the members of the set $C = \{v_i\} \cup \mathcal{P}(v_i)$ appearing in (10.10) are indeed neighbors of one another by $\mathcal{Q}$, as $\mathcal{Q}$ is now the same as $\mathcal{L}$, and in $\mathcal{L}$ all mates are neighbors (cf. (10.6)). It should also be noted that, in order to comply with our definition of potentials in Chapter 9, the set C in (10.10) should be allowed to be a proper subset of $\{v_i\} \cup \mathcal{P}(v_i)$ (always including v_i) whenever $C = \{v_i\} \cup \mathcal{P}(v_i)$ would yield a potential that is constant with respect to some $v_j \in \mathcal{P}(v_i)$. In this case, V_C should be a constant for $C = \{v_i\} \cup \mathcal{P}(v_i)$ and follow (10.10) for the largest subset of C that includes v_i yielding a nonconstant potential.

Theorem 10.3. *The set $V - \mathcal{E}$ is a GRF of energy*

$$E(d_1, \ldots, d_n) = \sum_{C \in \mathbf{C}} V_C(d_1, \ldots, d_n)$$
$$= -\sum_{i=1}^{n} \ln P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i))$$

with respect to $\mathcal{L}$ and P, for all $(d_1, \ldots, d_n) \in \{0,1\}^n$ and temperature $T = 1$.

Proof: This energy with $T = 1$ yields, by (9.4), a Boltzmann-Gibbs distribution π such that

$$\pi(d_i;\ v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \frac{\prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))}{\sum_{(d'_1, \ldots, d'_n) \in D^n} \prod_{j=1}^{n} P(d'_j \mid d'_k;\ v_k \in \mathcal{P}(v_j))},$$

where

$$\alpha = \frac{1}{\pi(d_j;\ v_j \in \mathcal{E})}$$

is a normalizing constant, as it does not depend on any d_i such that $v_i \in V - \mathcal{E}$. By (10.3), we then have

$$\pi(d_i;\ v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \frac{\prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j))}{\sum_{(d'_1, \ldots, d'_n) \in D^n} P(d'_1, \ldots, d'_n)}$$
$$= \alpha \prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

which is the joint distribution asserted by Theorem 10.1. ∎

It is interesting to check that the energy E of Theorem 10.3 yields (as discussed in the theorem's proof), by (9.4) with $T = 1$, a Boltzmann-Gibbs distribution on $V - \mathcal{E}$ that is the very distribution asserted in Theorem 10.1, i.e.,

$$\pi(d_i; v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

where α is a normalizing constant. In addition, the energy E_i that depends only on those subsets $C \in \mathbf{C}$ including v_i (cf. (9.6)) is by Theorem 10.3 given by

$$E_i(d_1, \ldots, d_n) = -\ln P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) - \sum_{v_j \in \mathcal{C}(v_i)} \ln P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

so it is also simple to check that the conditional probability in (9.5) translates into

$$P(d_i \mid d_j;\ v_j \neq v_i) = \alpha P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)),$$

which is the one asserted by Theorem 10.2 and rewritten in (10.9) to make the normalizing constant α explicit. As a consequence, performing stochastic simulation by applying (10.9) to the variables in $V - \mathcal{E}$ is indeed an instance of Gibbs sampling, and for this reason Theorems 9.2 through 9.4 apply. Theorem 9.2, in particular, ensures convergence to the distribution of Theorem 10.1 (we shall discuss the consequences of the other two theorems shortly).

10.2. SIMULATION ALGORITHMS

10.2.1. The sequential algorithm

As we discussed in Section 10.1, the process of stochastic simulation endows a Bayesian network with the characteristics of a GRF (hence an MRF, by Theorem 9.1) of temperature $T = 1$ with respect to the neighborhood $\mathcal{L}$ of (10.6) and the distribution P. A Bayesian network under stochastic simulation is then one more instance of a PC automaton network, as a consequence of our discussion in Section 9.2 concerning an MRF under Gibbs sampling. G is in this case obtained by letting $\mathcal{N}(n_i)$ be the set of nodes to which the variables in $\mathcal{L}(v_i)$ correspond. As in Section 9.2, if for some $v_j \in \mathcal{P}(v_i)$ V_C is constant for all C in which v_i and v_j appear, then n_j need not be included in $\mathcal{N}(n_i)$, even though $v_j \in \mathcal{L}(v_i)$. The graph G can be regarded as the extension, by adding undirected edges between mates, of the undirected graph that underlies the Bayesian network (Figure 10.1). The updating function f of the PC automaton network is such that $x_i(s)$ is given by the initial value of v_i for $s = 0$, and for $s > 0$ by the value of v_i after the most recent pulse in which n_i was in the independent set. Henceforth, d_i^0 denotes the initial value of v_i. This PC automaton network can be simulated by a sequential algorithm that scans the variables in V in a fixed order and updates them according to (10.9). Such an algorithm is given next as Algorithm *Seq_Bayesian*. It runs for a pre-specified number K of full scans, so every variable is updated exactly K times.

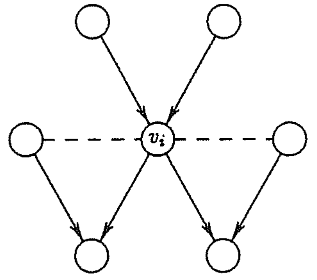


Figure 10.1. *Directed edges in the graph fragment belong to the Bayesian network. Dashed edges have been added between v_i and its mates to yield the corresponding fragment of G, in which the Bayesian-network edges should be regarded as undirected.*

```
Algorithm Seq_Bayesian:
    for $i := 1$ to $n$ do
        $v_i := d_i^0$;
    for $k := 1$ to $K$ do
        for $i := 1$ to $n$ do
            if $v_i \notin \mathcal{E}$ then
                begin
                    Let
                        $\rho_j^0 = P(d_j \mid d_i = 0, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j))$
                    and
                        $\rho_j^1 = P(d_j \mid d_i = 1, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j))$
                    for all $v_j \in \mathcal{C}(v_i)$;
                    Evaluate the conditional probability
                    $P(d_i \mid d_j;\ v_j \neq v_i)$
                    $= \frac{P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i}}{\sum_{d \in \{0,1\}} P(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d}}$
                    for all $d_i \in \{0,1\}$, and choose a value for $v_i$ accordingly
                end.
```

Note that variables in $\mathcal{E}$ are never updated, as they constitute the input to the simulation and must remain fixed all the time.

10.2.2. The distributed parallel algorithm

The approach of stochastic simulation characterizes a Bayesian network as a GRF of temperature $T = 1$ with respect to the neighborhood $\mathcal{L}$ of (10.6) and the distribution P. A Bayesian network can then be viewed as a PC automaton network, just as an MRF under Gibbs sampling (see Section 9.2).

The template introduced in Section 4.3.3 can then be used in coining a distributed parallel algorithm to simulate this PC automaton network. In this case, processor p_0 does not participate in detecting the termination of the simulation, as every variable is updated the same number K of times. Instead, p_0 participates by performing calculations to estimate the probabilities in (10.4), which as we know are the reason for conducting the stochastic simulation. We shall return to this point and give more details shortly. In order to fill the template for the distributed parallel simulation of PC automaton networks, the procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ must be specified for every processor p_i such that $n_i \in N$. Procedure UPDATE$_i$(MSG_i) is, as Algorithm *Seq_Bayesian* of Section 10.2.1, based on (10.9).

We know from Section 4.3.3 that the set MSG_i contains exactly one message from each of p_i's neighbors. As we anticipated in that section, in the case of Bayesian networks these messages are not necessarily all of the same type, in contrast with the other automaton networks we have encountered throughout the book, in which cases the messages invariably contained the states of the neighbors' corresponding nodes. In the case of a Bayesian network, each of these messages may be a variable's value, a pair of conditional probabilities, or a mere signal, as we explain next.

In our distributed parallel algorithm to perform the stochastic simulation of a Bayesian network, each processor p_i stores the conditional probabilities, given by (10.1), of v_i's values. As a first consequence of this data partitioning among the processors, there has to be a processor even for each of the variables in $\mathcal{E}$, although only variables in $V - \mathcal{E}$ are updated during the stochastic simulation (variables in $\mathcal{E}$ are the input to the simulation and retain their initial values throughout). Such a processor is responsible for conveying to each of the processors lodging its variable's parents the two conditional probabilities of the variable's present value, given the two possible values of that parent and the values of the other parents, for use as the ρ's that appear in (10.9) (cf. (10.7) and (10.8)). This same function is also performed by all other processors, except those whose variables have no parents. In order to apply (10.9), a processor also requires the values of its variable's parents, so to the processors lodging its variable's children (if any) a processor sends its variable's value. As a general rule, p_i expects to receive v_j's value from p_j if v_j is a parent of v_i, a pair of conditional probabilities if v_j is a child of v_i, and a mere signal if v_j is a mate of v_i (in this case, the impact of v_j's value will be received from p_k such that v_k is a child of both v_i and v_j). These are then the contents of MSG_i.

There is one problem, though, with this variety of message types. Initially, a processor can only send conditional probabilities to the processors that lodge its variable's parents after those variables' initial values have been received. In order to keep with the template for the distributed parallel simulation of PC automaton networks, we then assume that the initial value of a variable is available at the processor that lodges that variable and at the processors that lodge that variable's children. It should be noted, however, that this assumption implies that the sending of a variable's value to p_i's downstream neighbors in INITIALIZE$_i$ is a redundancy (one that we maintain, for compatibility with the overall template). With the same purpose of keeping with the template, we also assume that pairs of conditional probabilities are sent as a single message. The actions of p_i for simulating the PC automaton network corresponding to a Bayesian network are specified by the procedures that follow.

INITIALIZE$_i$:

1. Let $v_i := d_i^0$;
2. Send d_i to p_0;
3. For $v_j \in \mathcal{P}(v_i)$, let

$$\pi_j^0 = P\big(d_i \mid d_j = 0, d_k^0;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$$

and

$$\pi_j^1 = P\big(d_i \mid d_j = 1, d_k^0;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big).$$

Send (π_j^0, π_j^1) to every downstream neighbor p_j of p_i such that $v_j \in \mathcal{P}(v_i)$;

4. Send d_i to every downstream neighbor p_j of p_i such that $v_j \in \mathcal{C}(v_i)$;
5. Send a signal to every downstream neighbor p_j of p_i such that $v_j \in \mathcal{M}(v_i)$.

UPDATE$_i$(MSG_i):

1. Let $d_j \in MSG_i$ be the value of v_j for all $v_j \in \mathcal{P}(v_i)$. Let $(\rho_j^0, \rho_j^1) \in MSG_i$ be a pair of conditional probabilities stored at p_j for all $v_j \in \mathcal{C}(v_i)$;
2. If $v_i \notin \mathcal{E}$, then evaluate the conditional probability

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{P\big(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)\big) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i}}{\sum_{d \in \{0,1\}} P\big(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)\big) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d}}$$

for all $d_i \in \{0,1\}$, and choose a value for v_i accordingly.

NOTIFY$_i$:

1. Send d_i to p_0;
2. For $v_j \in \mathcal{P}(v_i)$, let

$$\pi_j^0 = P\big(d_i \mid d_j = 0, d_k;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$$

and

$$\pi_j^1 = P\big(d_i \mid d_j = 1, d_k;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big).$$

Send (π_j^0, π_j^1) to every p_j such that $v_j \in \mathcal{P}(v_i)$;

3. Send d_i to every p_j such that $v_j \in \mathcal{C}(v_i)$;
4. Send a signal to every p_j such that $v_j \in \mathcal{M}(v_i)$.

In these procedures, d_i is a variable local to p_i.

10.2.3. Two variations of stochastic simulation

Having described the main mechanism of stochastic simulation as implemented by a sequential and a distributed parallel algorithm in Sections 10.2.1 and 10.2.2, respectively, we are now in position to be more precise about the two variations of stochastic simulation alluded to earlier in Section 10.1.

The first variation estimates $P(d_i \mid d_j;\ v_j \in \mathcal{E})$ (cf. (10.4)) proportionally to the number of times v_i is assigned value d_i in Algorithm *Seq_Bayesian* or in step 2 of the procedure UPDATE$_i$. The second variation estimates this probability as the average of the probabilities evaluated in the inner loop of Algorithm *Seq_Bayesian* or in step 2 of the procedure UPDATE$_i$, i.e., those given by (10.9). In the case of the distributed parallel algorithm, these calculations are carried out by processor p_0. Note, however, that the second approach is not supported by the algorithm given in Section 10.2.2, and would require that every processor send to p_0 not only the values of its variable but also the conditional probabilities resulting from applying (10.9).

That these two approaches yield results consonant with the distribution to which the simulation converges (see Theorem 10.1) is an immediate consequence of Theorem 9.4 on the ergodicity of an MRF. To see this, in Theorem 9.4 let

$$g(V(k)) = d_i$$

with d_i being the ith coordinate of $V(k)$ to justify the first approach, and

$$g(V(k)) = P(d_i \mid d_j;\ v_j \neq v_i)$$

with $(d_1, \ldots, d_n) = V(k)$ to justify the second approach.

10.3. FURTHER PROPERTIES

Besides the problem of evaluating the conditional probabilities in (10.4) for variables in $V - \mathcal{E}$, another important problem of relevance within the context of Bayesian networks is to identify a point in $\{0,1\}^{|V-\mathcal{E}|}$ at which the joint probability distribution asserted in Theorem 10.1 is maximum. Whereas the solutions to the former problem are based on a view of a Bayesian network as a GRF with $T = 1$ (see Section 10.1 and Section 10.2.3), the latter problem can be approached by considering GRF's with $T \neq 1$.

Suppose we take the energy given in Theorem 10.3,

$$E(d_1, \ldots, d_n) = -\sum_{i=1}^{n} \ln P\big(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)\big),$$

and write the corresponding Boltzmann-Gibbs distribution on $V - \mathcal{E}$ with $T \neq 1$. This distribution is, by (9.4), given by

$$\pi(d_i; v_i \in V - \mathcal{E} \mid d_j;\ v_j \in \mathcal{E}) = \alpha \left(\prod_{j=1}^{n} P(d_j \mid d_k;\ v_k \in \mathcal{P}(v_j)) \right)^{1/T}, \qquad (10.11)$$

where α is a normalizing constant. If we perform Gibbs sampling on the GRF whose distribution is given by (10.11), then by Theorem 9.3 we can reach a global minimum of the energy of Theorem 10.3, as long as the constraints given in that theorem are not violated. Such a global minimum is of course a global maximum of

$$\prod_{i=1}^{n} P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)),$$

and then by Theorem 10.1 occurs at the most likely point in $\{0,1\}^{|V-\mathcal{E}|}$, given the values of the variables in $\mathcal{E}$.

By Theorem 9.3, such a maximum-likelihood point can be reached by performing Gibbs sampling with varying temperature T, as in Section 9.3.2. This is a variation of the stochastic simulation procedure of Section 10.1 (in fact, a combination of that procedure with the simulated annealing method discussed in Section 9.3.1), and employs the conditional probabilities derived from (10.11),

$$P(d_i \mid d_j;\ v_j \neq v_i) = \frac{\left(P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i} \right)^{1/T}}{\sum_{d \in \{0,1\}} \left(P(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d} \right)^{1/T}}, \qquad (10.12)$$

for all $v_i \in \{V - \mathcal{E}\}$. The probabilities in (10.12) can be regarded as the ones in (10.9) with the probabilities conditioned on parents' values raised to the $(1/T)$th power. Along this T-dependent variation of stochastic simulation, T is continually decreased from an initial high value. As in Section 9.3.2, the cooling schedule given in Theorem 9.3 cannot truly be obeyed. The more realistic schedule of (9.8) can then be adopted, in which case each variable is updated the number of times given in (9.9) before a pre-established final value of T is reached. The result is then approximate, as the guarantee of convergence to the optimum given by Theorem 9.3 is lost.

10.4. FURTHER SIMULATION ALGORITHMS

10.4.1. The sequential algorithm

Following our discussion in Section 10.3, the process of T-dependent stochastic simulation of a Bayesian network characterizes the network as a GRF (hence an MRF, by Theorem 9.1) with varying temperature T. As we discussed in Section 9.4.1, an

MRF in such circumstances can still be regarded as giving rise to an instance of a PC automaton network, and then so can the Bayesian network. This PC automaton network is identical to the one described in Section 10.2.1, except for the updating function f, which now depends on a (varying) temperature T. A sequential algorithm to simulate this PC automaton network employing the approximate cooling schedule of (9.8) is given next as Algorithm *Seq_Annealed_Bayesian*. This algorithm employs (10.12) for variable updating.

```
Algorithm Seq_Annealed_Bayesian:
    for $i := 1$ to $n$ do
        $v_i := d_i^0$;
    $T := T_0$;
    while $T \geq T_f$ do
        begin
            for $i := 1$ to $n$ do
                if $v_i \notin \mathcal{E}$ then
                    begin
                        Let
                            $\rho_j^0 = P(d_j \mid d_i = 0, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j))$
                        and
                            $\rho_j^1 = P(d_j \mid d_i = 1, d_k;\ v_k \neq v_i, v_k \in \mathcal{P}(v_j))$
                        for all $v_j \in \mathcal{C}(v_i)$;
                        Evaluate the conditional probability
                $P(d_i \mid d_j;\ v_j \neq v_i)$
                $= \frac{\left( P(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i} \right)^{1/T}}{\sum_{d \in \{0,1\}} \left( P(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d} \right)^{1/T}}$
                        for all $d_i \in \{0,1\}$, and choose a value for $v_i$ accordingly
                    end;
            $T := \alpha T$
        end.
```

As in Algorithm *Seq_Bayesian*, variables in $\mathcal{E}$ are never updated. Each of the other variables is updated a number of times given by (9.9).

10.4.2. The distributed parallel algorithm

Algorithm *Seq_Annealed_Bayesian* has a distributed parallel counterpart that can be obtained rather directly from the distributed parallel algorithm given in Section 10.2.2 for stochastic simulation. The main steps of the three procedures INITIALIZE$_i$, UPDATE$_i$(MSG_i), and NOTIFY$_i$ for processor p_i are given next. As in previous situations, MSG_i contains exactly one message from each of p_i's neighbors, in this case given as explained in Section 10.2.2: the values of its variable's parents, from the processors that lodge its variable's parents, one pair of probabilities from each of the processors lodging its variable's children, and a signal from each of the processors lodging its variable's mates. Also as in Section 10.2.2, a variable's initial value is assumed to be initially available to the processor that lodges it and to the processors that lodge its children. As before, processor p_0 has no participation in the detection of the simulation's termination, as each variable is updated a fixed number of times given by (9.9).

```
INITIALIZE$_i$:
 1. Let $v_i := d_i^0$;
 2. Send $d_i$ to $p_0$;
 3. For $v_j \in \mathcal{P}(v_i)$, let
        $\pi_j^0 = P\big(d_i \mid d_j = 0, d_k^0;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$
    and
        $\pi_j^1 = P\big(d_i \mid d_j = 1, d_k^0;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$.
    Send $(\pi_j^0, \pi_j^1)$ to every downstream neighbor $p_j$ of $p_i$ such that $v_j \in \mathcal{P}(v_i)$;
 4. Send $d_i$ to every downstream neighbor $p_j$ of $p_i$ such that $v_j \in \mathcal{C}(v_i)$;
 5. Send a signal to every downstream neighbor $p_j$ of $p_i$ such that $v_j \in \mathcal{M}(v_i)$;
 6. $T := T_0$.
UPDATE$_i$($MSG_i$):
 1. Let $d_j \in MSG_i$ be the value of $v_j$ for all $v_j \in \mathcal{P}(v_i)$. Let $(\rho_j^0, \rho_j^1) \in MSG_i$
    be a pair of conditional probabilities stored at $p_j$ for all $v_j \in \mathcal{C}(v_i)$;
 2. If $v_i \notin \mathcal{E}$, then evaluate the conditional probability
        $P(d_i \mid d_j;\ v_j \neq v_i)$
            $= \frac{\left(P\big(d_i \mid d_j;\ v_j \in \mathcal{P}(v_i)\big) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d_i}\right)^{1/T}}{\sum_{d \in \{0,1\}} \left(P\big(d_i = d \mid d_j;\ v_j \in \mathcal{P}(v_i)\big) \prod_{v_j \in \mathcal{C}(v_i)} \rho_j^{d}\right)^{1/T}}$
    for all $d_i \in \{0,1\}$, and choose a value for $v_i$ accordingly;
 3. $T := \alpha T$.
NOTIFY$_i$:
 1. Send $d_i$ to $p_0$;
 2. For $v_j \in \mathcal{P}(v_i)$, let
        $\pi_j^0 = P\big(d_i \mid d_j = 0, d_k;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$
    and
        $\pi_j^1 = P\big(d_i \mid d_j = 1, d_k;\ v_k \neq v_j, v_k \in \mathcal{P}(v_i)\big)$.
    Send $(\pi_j^0, \pi_j^1)$ to every $p_j$ such that $v_j \in \mathcal{P}(v_i)$;
 3. Send $d_i$ to every $p_j$ such that $v_j \in \mathcal{C}(v_i)$;
 4. Send a signal to every $p_j$ such that $v_j \in \mathcal{M}(v_i)$.
```

In these procedures, d_i is a variable local to p_i.

10.5. LEXICAL DISAMBIGUATION

Bayesian networks are ideally suited to the modeling of evidential reasoning in areas in which the interplay between concepts follows the pattern that exists, for example, in applications within the medical sciences. In such fields, what one seeks is essentially the probability of a certain disease given a set of symptoms. What one has to begin with, on the other hand, are the probabilities of each symptom, given the various combinations of the diseases believed to be direct causes of that symptom. In other domains in which the roles as causes and effects are not so well defined, Bayesian networks can still be used, and can sometimes elicit solutions within the uncertain domain in a manner that other approaches would hardly be able to.

In this section, we address the problem of resolving ambiguities within the processing of natural languages. This is a central problem in the field, and has received considerable attention throughout the years. As in practically all of natural language processing, the classical approach to the disambiguation of natural sentences is essentially algorithmic and tailored to the execution by traditional sequential computers. More recently, however, various approaches based on neural networks have been proposed, both for the very strong appeal of parallel computers and for the (surely less agreeable) plausibility of brain function imitation. The resulting proposals have been termed *connectionist*, in allusion to the role played by the synaptic strengths of the connections between neurons in the collective behavior of neural networks. The neural networks used in these proposals are different from the ones we considered in Chapters 6 through 8, and frequently must incorporate mechanisms, extraneous to the neural model and typically handled rather informally, to ensure proper functioning.

Ambiguities of various types occur in natural sentences. *Structural ambiguities*, for example, are related to how the phrases within the sentence are attached to one another for interpretation, as in the case of *I saw the man on the hill with a telescope*. Clearly, the ambiguity in this example cannot be resolved without additional contextual information. *Lexical ambiguities*, on the other hand, are the ambiguities of grammatical categories and word senses, as for example in *Time flies like an arrow*. Although in this case the sentence's verb might in principle be any of *Time*, *flies*, or *like*, we have no difficulty in identifying *flies* as the only real possibility, as the required complements to the other candidates are not fully present in the sentence. The complements to *flies*, though, are all there, as *Time* is the verb's "agent" and *arrow* a "modifier." These denominations of a verb's complements are borrowed from the linguistic notions of the *cases* of verbs, and have been employed relatively informally in the resolution of ambiguities, as in some of the most prominent connectionist approaches. Verb cases specify the complements that are required for each of the intended meanings of the verb. Various possibilities exist, as an *agent*, an *object*, a *modifier*, etc. In the remainder of this section, we deal exclusively with the resolution of lexical ambiguities.

We have seen in the preceding discussion that the interplay between the syntactic and semantic characteristics of a sentence's components is crucial to the

resolution of lexical ambiguities. In particular, if the disambiguation process is not preferentially directed by neither the syntax nor the semantics, but rather by the two cooperating fronts concomitantly, then one can as an advantage aim at interpreting sentences like *fire arson match hotel*, where the absence of a grammatically correct structure only slightly disturbs one's understanding of the sentence. If the interpretation of sentences can be carried out within a model that allows diagnostics to be issued with an attached measure of likelihood, then the possibilities for treating sentences lacking a rigid grammatical structure are additionally enhanced. Bayesian networks, being models with such a characteristic, seem then naturally suited to constitute the core of an approach to the resolution of lexical ambiguities.

Let us then review the main characteristics of a Bayesian network to tackle the problem of lexical ambiguity resolution. The account we give here is superficial, and additional details can be looked up in the literature. The first important characteristic is probably inherent to any model of natural language processing that operates in parallel, and has to do with the length (number of words) of the sentences that can be accepted as input. As all the words in the sentence are presented concurrently to the network, the length of the sentence is limited, and is here bounded by a positive integer that we denote by ℓ. Similarly, the maximum number of distinct words that the network can handle must also be specified, and is denoted by L. The set $\mathcal{E}$ of evidences can be regarded as being organized in two two-dimensional arrays, both with ℓ columns. The first array has one row for each possible grammatical category of a word and the second array has L rows. When a sentence of length $l \leq \ell$ is input to the network, the variables occupying columns $l+1, \ldots, \ell$ of both arrays are set to 0. For $s \leq l$, a variable on column s and row r of the first array is set to 1 if and only if one of the possible grammatical categories of the sth word in the sentence is the grammatical category corresponding to row r. A column in this array may then contain more than one variable set to 1. The same column in the second array, on the other hand, contains exactly one variable set to 1, and this variable is on the row that corresponds to the word that occupies the sth position in the sentence.

The entire network comprises two subnetworks, one devoted to the syntactic analysis of the sentence, the other devoted to the semantic analysis of the sentence. The first array of evidences is then the input to the subnetwork for syntactic analysis, and the second array is the input to the subnetwork for semantic analysis. The two subnetworks are interconnected, and cooperate with each other in the process of lexical disambiguation.

The subnetwork for syntactic analysis is derived from a context-free grammar, which summarizes the various possibilities for syntactically correct sentences. This grammar must contain special productions in which the various combinations of verbs, noun phrases (*NP*), and prepositional phrases (*PP*) of relevance to determine the cases of each verb are made explicit. For example, the grammar $\mathcal{G}$ whose starting symbol is S and whose productions are

1. $S \rightarrow W\ PP$
2. $S \rightarrow W\ NP$
3. $S \rightarrow W\ NP\ PP$
4. $W \rightarrow NP\ verb$
5. $W \rightarrow verb$
6. $PP \rightarrow prep\ NP$
7. $NP \rightarrow det\ N$
8. $NP \rightarrow N$
9. $N \rightarrow adj\ N$
10. $N \rightarrow noun$

has been devised in such a manner that the first three productions indicate the possibilities that matter in the coarse structure of some sentences concerning the determination of the verb's cases. In the grammar $\mathcal{G}$, the terminal symbols *verb*, *prep*, *det*, *adj*, and *noun* stand for the possible grammatical categories of each word. According to $\mathcal{G}$, the sentence *Time flies like an arrow* can be parsed in three different ways, corresponding to selecting *Time*, *flies*, or *like* as *verb* (Figure 10.2). The subnetwork derived from $\mathcal{G}$ can be regarded as a collection of parse trees that have been coalesced to eliminate duplicate nodes. Its edges are directed toward the trees' roots, and then the evidences are variables without parents (their prior probabilities are irrelevant, though, as they are evidences and as such their values never change). The trees' roots correspond to the productions that have S on the left-hand side (Figure 10.3), and are used to interconnect the syntactic and semantic subnetworks. In the syntactic subnetwork, the probabilities of variables' values, conditioned on parents' values (cf. (10.1)), are set in a rather direct way, following the productions in $\mathcal{G}$.

In order to describe the subnetwork for semantic analysis, we draw on the example sentence *Time flies like an arrow*. For this sentence, let S_1, S_2, and S_3 be the variables at the parse trees' roots, corresponding respectively to taking productions 1, 2, and 3 in the first place (i.e., corresponding respectively to taking *flies*, *like*, and *Time* as *verb*). The parent set of each evidence to the semantic subnetwork comprises these three variables, in addition to variables that describe the various possibilities of the corresponding word as *verb* or as part of an *NP*. For example, if *flies*2 is the evidence corresponding to the word *flies* in the second position in the sentence, then

$$\mathcal{P}(\mathit{flies}2) = \{S_1, S_2, S_3, \mathit{fliesNP}0, \mathit{fliesV}, \mathit{fliesNP}1\},$$

where *fliesNP*0 indicates the possibility of having *flies* in an *NP* before *verb*, *fliesV* of having it as *verb*, and *fliesNP*1 in the first *NP* after *verb*. Similarly for the other evidences to the semantic subnetwork (Figure 10.4).

The conditional probabilities associated with these evidences (cf. (10.1)) indicate the possible roles for each word, depending on the parsing diagnostics represented by S_1, S_2, and S_3. In the case of *flies*2, the only conditional probabilities with significant values are

$$P(\mathit{flies}2 = 1 \mid \mathit{fliesNP}0 = s_1, \mathit{fliesV} = 1, \mathit{fliesNP}1 = s_2, S_1 = 1, S_2 = 0, S_3 = 0), \tag{10.13}$$

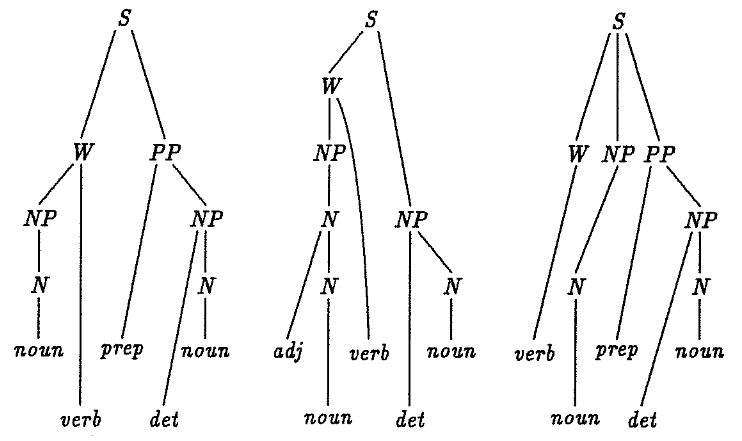


Figure 10.2. *These are the three parse trees for Time flies like an arrow according to $\mathcal{G}$. They correspond to a choice of flies, like, or Time as verb.*

$$P(\mathit{flies}2 = 1 \mid \mathit{fliesNP}0 = 1, \mathit{fliesV} = s_1, \mathit{fliesNP}1 = s_2, S_1 = 0, S_2 = 1, S_3 = 0), \tag{10.14}$$

and

$$P(\mathit{flies}2 = 1 \mid \mathit{fliesNP}0 = s_1, \mathit{fliesV} = s_2, \mathit{fliesNP}1 = 1, S_1 = 0, S_2 = 0, S_3 = 1), \tag{10.15}$$

where $s_1, s_2 \in \{0, 1\}$ (the notation in (10.13) through (10.15) is in slight disaccord with our previous notation, as we have now used, for notational simplicity, a variable's name for its value). In (10.13), for example, we state that *flies*2 = 1 with significant probability, given that *fliesV* = 1, $S_1 = 1$, and $S_2 = S_3 = 0$, regardless of the values of *fliesNP*0 and *fliesNP*1. In other words, in order for *flies*2 to be the sentence's verb it suffices that at least one parsing diagnostic in which *verb* appears in the second position be present (S_1, in this case), while those that do not have *verb* in that position be all absent (S_2 and S_3, in this case). It is important to note that all combinations of *fliesNP*0 and *fliesNP*1 should be allowed in conjunction with *fliesV* = S_1 = 1, since simply setting *fliesNP*0 = *fliesNP*1 = 0 would forbid the occurrence of sentences like *Flies fly*, *Men man*, etc., where basically the same word appears in different positions.

Similarly, the conditional probability in (10.14) indicates that *flies*2 is in an *NP* before *verb* if at least one parsing diagnostic with *verb* in the third position (or farther to the right) is present (S_2, in this case), while those where *verb* is not in the third position (or farther to the right) are all absent (S_1 and S_3, in this case). This

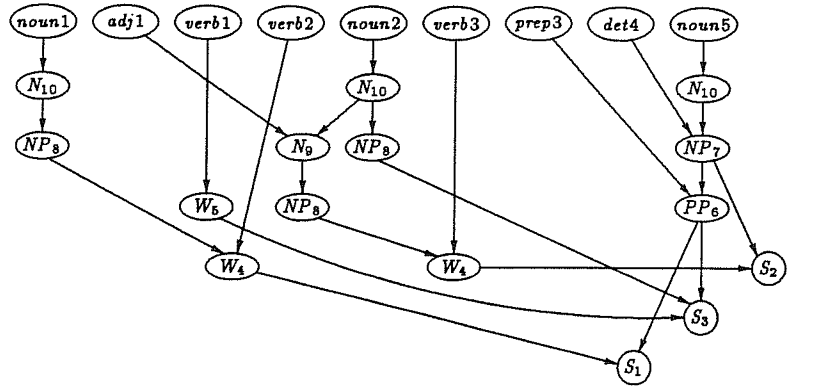


Figure 10.3. *In this figure, a portion of the Bayesian network for parsing sentences in the language generated by $\mathcal{G}$ is shown. This portion corresponds to the three possible ways of parsing Time flies like an arrow. Evidences are the variables without any parents, and correspond to the various possible grammatical categories in each position of the sentence. So noun*1, *for example, indicates the possibility of a noun in the first position. In the identification of each of the other variables, the subscript corresponds to the production in $\mathcal{G}$ to which that variable corresponds.*

same reasoning applies to the probability in (10.15) and to all the other evidences of the semantic subnetwork.

Aside from S_1, S_2, and S_3, which convey information about the sentence's parse tree, additional variables to look at for a complete syntactic diagnostic (relating each word to its phrase within the sentence) are the other variables in the parent sets of the evidences to the semantic subnetwork. These variables indicate the most likely candidate for *verb* (*fliesV*, in this case), along with the accompanying *NP*'s before and after *verb* (*TimeNP*0 and *arrowNP*1, respectively, in this case). These variables in turn have as parents variables related to the various possible word senses of the candidates for the three *NP*'s and for *verb*. Networks for case hierarchies of the *NP*'s are then used to fulfill the required cases of the verbs (Figure 10.4). For example, the variable *TimeV* has as a parent the variable $T1$, which represents the possibility of having *Time* used as a verb in the sense of "timing athletic events," or "timing the arrival of airplanes," etc. Node *fliesV* in its turn has two parents, $F1$ and $F2$, representing respectively the use of *flies* as a verb in the sense of "time elapsing" or of "piloting an aircraft." Each of these variables for verb senses has as parents variables for the corresponding cases. For example, $F1$ has a parent $F1obj$ corresponding to a required object (Figure 10.5).

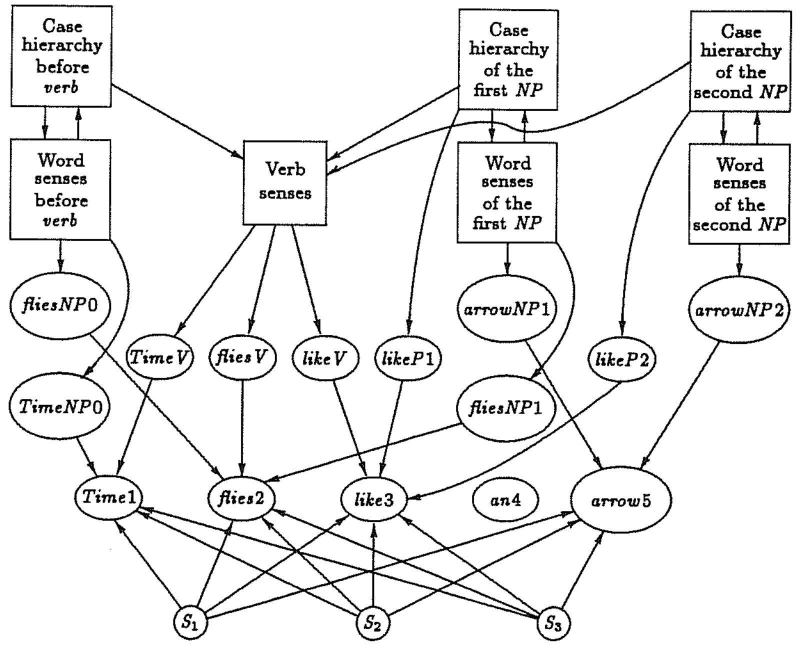

Case hierarchy before verb
Case hierarchy of the first NP
Case hierarchy of the second NP
Word senses before verb
Verb senses
Word senses of the first NP
Word senses of the second NP
fliesNP0
arrowNP1
arrowNP2
TimeV
fliesV
likeV
likeP1
likeP2
TimeNP0
fliesNP1
Time1
flies2
like3
an4
arrow5
S_1
S_2
S_3

Figure 10.4. *An overview is shown here of the Bayesian network for lexical disambiguation, omitting however most of its portion for parsing. Evidences for the semantic portion are the words occurring in each of the positions of the sentence. For example, Time*1 *indicates the occurrence of Time in the first position. The evidences in this case have been set for the sentence Time flies like an arrow. The remainder of the variables correspond to word senses and case hierarchies for the sentence's components.*

The semantic diagnostic is given by the network in the word-sense variables and in the variables that specify verb cases. For *Time flies like an arrow*, a highly likely conjunction of events in which the variables *Time period* (a parent of *TimeNP*0), $F1$, and $F1obj$ have value 1 is part of the correct diagnostic (Figure 10.5). That this diagnostic can indeed be expected to be achieved comes from the realization that only the sense corresponding to variable $F1$ will have its cases completely fulfilled. As a consequence, *fliesV* occurs with much higher probability than *TimeV*, and by the conditional probabilities of the variables *Time*1 and *flies*2, *TimeNP*0 occurs with much higher probability than *fliesNP*0.

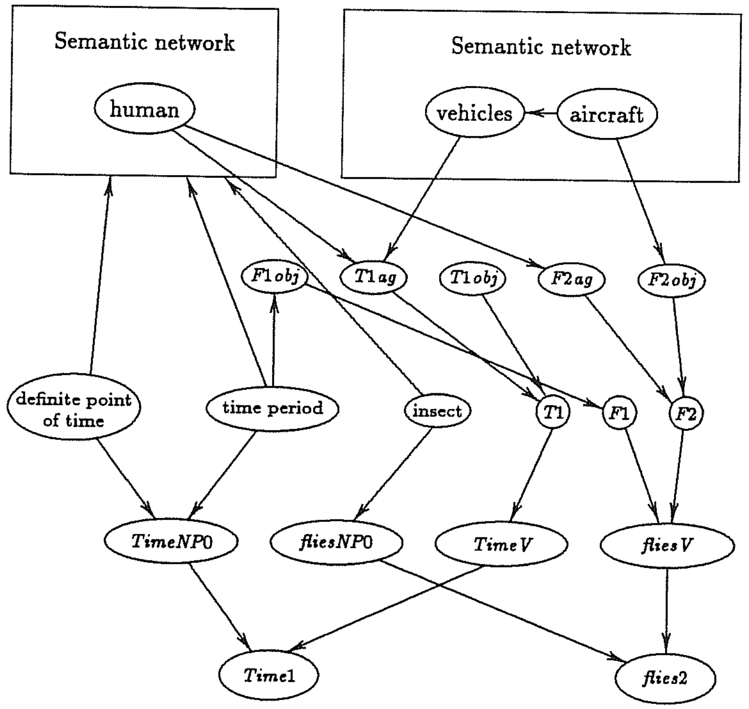


Figure 10.5. *This is a portion of the Bayesian network for semantic analysis. What is shown in this figure are details of a part of Figure 10.4, emphasizing some variables for word sense and for the case hierarchies. Topmost in this figure are semantic networks, used for the mechanism of case fulfillment.*

The variables for word-sense resolution and for case fulfillment are connected to semantic networks comprising variables for a Bayesian representation of knowledge (Figure 10.5). This topic is, however, still subject to considerable ongoing debate.

Once the Bayesian network has been completely specified and the evidences have been fixed, looking for the most likely point in $\{0,1\}^{|V-\mathcal{E}|}$ as discussed in Section 10.3 yields the desired diagnostic, embodying both the syntactic and the semantic diagnostics.

10.6. BIBLIOGRAPHIC NOTES

The reference for a great portion of the material presented in Section 10.1 is Pearl (1988). This includes Theorems 10.1 and 10.2, which had already appeared in Pearl (1986) and Pearl (1987), respectively. The relation between the structure of the directed graph corresponding to the Bayesian network and the independencies dictated by the probability distribution is expressed by the so-called *d*-separation criterion, which can also be found in Pearl (1988) and in Geiger, Verma, and Pearl (1990). The intractability of solving Bayesian networks exactly is the subject of Cooper (1990). The approach of stochastic simulation is due to Pearl (1987). Other approximate methods exist, and have been partly motivated by the difficulty of stochastic simulation to deal with extreme values of conditional probabilities. For a survey of some recent proposals, the reader should refer to Henrion (1990), and the references therein. The connection between the stochastic simulation of Bayesian networks and Gibbs sampling is based on Hrycej (1990).

The use of scheduling by edge reversal that underlies our algorithm for the distributed parallel simulation of PC automaton networks was already recommended in Pearl (1988) for the stochastic simulation of Bayesian networks by a parallel machine. This reference may be looked up as another source of part of the material in Section 10.2.

Section 10.3 on the variation of stochastic simulation that identifies the most likely joint assignment of values to the variables in a Bayesian network with the aid of simulated annealing is based on Hrycej (1990), where the connection between stochastic simulation and Gibbs sampling is extended to the variable-temperature case.

The application of Bayesian networks to the resolution of lexical ambiguities in the processing of natural languages, discussed in Section 10.5, is based on Eizirik (1990) and Eizirik, Barbosa, and Mendes (1993). References on the traditional approaches to natural language processing are Charniak (1983), Winograd (1983), and Allen (1987). Accounts on connectionist approaches have been given by Cottrell (1985), Fanty (1985), Selman (1985), Waltz and Pollack (1985), and McClelland and Kawamoto (1986). Relatively recent material on lexical ambiguity resolution can be found in Small, Cottrell, and Tanenhaus (1988). Another probabilistic approach that also utilizes Bayesian networks has been described by Charniak and Goldman (1989) (see also the more recent account in Goldman and Charniak (1992)). The references for verb cases are Fillmore (1968) and Samlowski (1976). Initial investigations into the probability-based representation of knowledge can be found in Bacchus (1988), Pearl (1989), and Bacchus (1991).

Part 5

Appendices

This part contains appendices, where material too specialized or too detailed to be included in the preceding chapters is presented.

Four appendices (Appendices A through D) are contained in Part 5. Appendix A is a primer on distributed parallel programming, and takes a more practical view when contrasted with the material of Part 2. Appendix B continues the material in Appendix A by providing an example of how the distributed parallel simulators studied throughout the book would be coded in the Occam programming language. Two long theorem proofs are provided in Appendix C, while Appendix D contains a discussion of additional properties of the basic distributed algorithm employed for the distributed parallel simulation of PC automaton networks.

A

A distributed parallel programming primer

In this appendix, we discuss the most important techniques needed to obtain a distributed parallel program from the abstract distributed algorithms discussed throughout the book. We begin with some concepts and notations in Section A.1, then proceed to a discussion of message buffering and task allocation, respectively in Sections A.2 and A.3. The issue of communication among processors is treated in Section A.4, which is followed by bibliographic notes in Section A.5.

A.1. DISTRIBUTED PARALLEL PROGRAMS

Throughout the book, we have adopted the graph $G = (N, E)$ as a descriptor of how the various parallel processing agents interact with one another. With each member n_i of N we have associated a processor p_i, and with each undirected edge (n_i, n_j) in E a bidirectional communication channel connecting processors p_i and p_j. All the algorithms we presented in the book were given by specifying, for each processor p_i, an atomic action to be performed by p_i upon receiving an external input, typically a message from one of its neighbors according to the structure of G. Performing this action has at times been conditioned upon additional occurrences, other than simply the reception of a message, as for example in the case of Algorithm *Gsr* of Section 3.2.2. An additional processor, p_0, has been employed in some situations, as in Sections 3.3 and 4.3, notably for global initiation and termination detection.

In this appendix and in Appendix B, we shall refer to the processors $p_0, \ldots, p_n$ as *tasks*, as the denomination as a *processor* will be used for a processor in a parallel machine (cf. Sections A.3 and A.4). We might have used *process* instead, but have refrained from doing so to avoid confusion with the concept of an Occam process, which is far more encompassing (cf. Section B.1). In fact, the set of tasks and their interconnecting channels will be represented by a directed graph

$G_T = (N_T, E_T)$, where N_T is the set of tasks and E_T is a set of unidirectional communication channels. In order to represent the graph $G = (N, E)$ used throughout the book, we would then have $N_T = N$ and E_T comprising two directed edges for each $(n_i, n_j) \in E$, one leading from n_i to n_j, the other from n_j to n_i. This representation might then be enlarged to include the additional task p_0 by simply adding p_0 to N_T and two additional channels to E_T for each of the tasks $p_1, \ldots, p_n$, one leading to p_0, the other leading from p_0.

The input/action pairs that characterize the behavior of a task $t \in N_T$ are usually expressed in the form of guarded commands (explained shortly), and then grouped together in a repetitive construct that ends when the condition of global termination is known to t. This local detection at t of a condition of global termination involves, in fact, more input/action pairs than the ones given throughout the book for most of the algorithms, as further communication is usually necessary for the information that global termination has been detected to be spread out through the system. We have given in Section 4.3 an account of how this takes place in the algorithms for automaton network simulation.

For a task $t \in N_T$, let *In*(t) and *Out*(t) denote, respectively, the subsets of E_T corresponding to channels that income to and outgo from t in G_T. A *guarded command* has two components, the *guard* part, and the *command* part. The guard involves the reception of a message by task t on one of the channels in *In*(t) and possibly a boolean condition B. A guarded command is then usually denoted by

$$guard \rightarrow command,$$

or, equivalently,

receive message on $c \in In(t)$ **and** $B \rightarrow$ *command*,

where **receive** is a generic language construct for receiving messages. A guard is said to be *ready* when a message is available for immediate reception on the specified channel and B is true.

The overall behavior of a task t can then be summarized along the lines of the algorithm we provide next, Algorithm *Task*(t).

Algorithm *Task*(t):

```
send one message on each channel
    of a (possibly empty) subset of $Out(t)$;
repeat
    receive message on $c_1 \in In(t)$ and $B_1 \rightarrow$
        do some computation;
        send one message on each channel
            of a (possibly empty) subset of $Out(t)$
    or...
    or
    receive message on $c_{n(t)} \in In(t)$ and $B_{n(t)} \rightarrow$
        do some computation;
        send one message on each channel
            of a (possibly empty) subset of $Out(t)$
until global termination is known to $t$.
```

In Algorithm *Task*(t), where $n(t) = |In(t)|$, the $n(t)$ guarded commands are grouped by connectives **or** and inside a **repeat...until** pair that is executed until the condition of global termination is known to t. At each iteration of the loop, exactly one of the guarded commands whose guards are ready is selected for execution. If more than one guard is ready the selection is arbitrary, whereas if none of the guards is ready the execution is halted until one is. Just like **receive**, **send** can be regarded as a generic language construct for sending messages. Whenever the (possibly empty) subset of *Out*(t) has more than one member, the messages are sent in parallel on these channels.

The use of **receive** in a guard should be interpreted as an indication that a message must be available for immediate consumption on the corresponding input channel for the guard to be ready. When used in a different context, **receive** usually exhibits a *blocking* nature, that is, the task that executes a **receive** is suspended until a message arrives to be received on the channel specified, unless a message is already there for reception, in which case it is received and the task resumes its execution.

The **send** construct also has a semantics of its own, and this is in general one of two possibilities. The **send** may be a *blocking* operation or an *nonblocking* operation. In the former case, the task is suspended until the message can be sent and received immediately by the task at the other end of the channel. This form of **send** then requires that the communicating tasks synchronize with each other for the exchange of information (this is a *task rendez-vous*), and is one of the bases of the CSP (Communicating Sequential Processes) notation for concurrent programming. In the nonblocking case, the task transmits the message and immediately resumes its execution. Naturally, this form of **send** requires buffering for the messages that have been sent but not received, i.e., messages that are in transit on a channel. Blocking and nonblocking **send** operations are also sometimes referred to as *synchronous* and *asynchronous*, respectively, in view of the end-task synchronization effect it has in the former case. We do, however, refrain from utilizing this nomenclature in the context of this book, as the properties of synchronism and asynchronism already have an established and important meaning (cf. Section 3.1.2).

The relation of blocking and nonblocking **send** operations with message buffering requirements raises important questions related to the design of distributed parallel algorithms. If, on the one hand, a blocking **send** requires no message buffering (as the message is passed directly between the synchronized tasks), on the other hand a nonblocking **send** requires the ability of a channel to buffer an unbounded number of messages. The former scenario poses great difficulties to the program designer, as communication deadlocks occur with great ease when the programming is done with the use of blocking constructs only. For this reason, however unreal the requirement of infinitely many buffers may seem, it is customary to start the design of a distributed parallel algorithm by assuming nonblocking constructs, and then at a later stage performing changes to yield a program that makes use of the constructs provided by the language at hand, possibly of a blocking nature (as in the case of the CSP-inspired Occam, used in Appendix B), or of a nature that lies somewhere in between the two extremes of blocking and nonblocking **send** operations (discussed in Section A.2).

Besides the increased ease in the design phase of a distributed algorithm (or perhaps because of it), the use of nonblocking constructs does in general allow the correctness of distributed algorithms to be shown more easily, as well as their properties. This justifies the first of the following two assumptions regarding Algorithm *Task*(t).

- The **send** operations have a nonblocking nature.
- Every channel $c \in C_T$ delivers messages in the FIFO order.

The second assumption is sometimes essential (cf. Theorem 3.2), but in general what it does is to simplify the process of distributed algorithm design, as we noted in Section 4.1.

A.2. MESSAGE BUFFERING

As we saw in Section A.1, the blocking or nonblocking nature of the **send** operations is closely related to the channels' ability to buffer messages. Specifically, blocking operations require no buffering at all, while nonblocking operations may require an infinite amount of buffers. Between the two extremes, we say that a channel has *capacity* $k \geq 0$ if the number of messages it can buffer before either a message is received by the receiver or the sender is blocked upon attempting a transmission is k. The case of $k = 0$ corresponds to a blocking **send**, and the case in which $k \to \infty$ corresponds to a nonblocking **send**.

Although Algorithm *Task*(t) of Section A.1 is written under the assumption of infinite-capacity channels, such an assumption is unreasonable, and must be dealt with somewhere along the programming process. This is in general achieved along two main steps. First, for each channel $c \in E_T$ a nonnegative integer $b(c)$ must be

determined that reflects the number of buffers actually needed by channel c. This number must be selected carefully, as an improper choice may introduce communication deadlocks in the program. Such a deadlock is represented by a directed cycle of tasks, all of which are suspended to send a message on the channel on the cycle, which cannot be done because all channels have been assigned insufficient storage space. Secondly, once the $b(c)$'s have been determined, Algorithm *Task*(t) must be changed so that it now employs **send** operations that can deal with the new channel capacities. Depending on the programming language at hand, this can be achieved rather easily. For example, in the Occam programming language, in which all channels have zero capacity, each channel $c \in E_T$ may be replaced with a serial arrangement of $b(c)$ *relay tasks* alternating with $b(c)+1$ zero-capacity channels. Each relay task has one input channel and one output channel, and has the sole function of sending on its output channel whatever it receives on its input channel. It has, in addition, a storage capacity of exactly one message, so the entire arrangement can be viewed as a $b(c)$-capacity channel. The construction of such positive-capacity channels in Occam is discussed in more detail in Section B.1. Other programming languages and environments may require interactions with the operating system in order to perform this assignment of bounded capacities to channels.

The real problem is of course to determine values for the $b(c)$'s in such a way that no new deadlock is introduced in the distributed algorithm (put more optimistically, the task is to ensure the deadlock-freedom of an originally deadlock-free program). In the remainder of this section, we describe solutions to this problem which are based on the availability of a bound $r(c)$, provided for each channel $c \in E_T$, on the number of messages that may require buffering in c when c has infinite capacity. This number $r(c)$ is the largest number of messages that will ever be in transit on c when the end task of c is itself attempting a message transmission, so the messages in transit have to be buffered.

Determining the $r(c)$'s can be very simple for some distributed algorithms, and can be very hard for others. For example, in the case of Algorithm *Sas* of Section 4.1.2 it is easy to see that $r(c) = 1$ for all $c \in E_T$, while for Algorithm *Ser* of Section 4.2.2 we have $b(c) = 0$ for all $c \in E_T$. The former is true because in Algorithm *Sas* two messages can only be in transit if the end task is expecting messages (including the first one of these two) in order to proceed to a further clock pulse in the synchronous algorithm (it cannot therefore be engaged in any message transmission, as this would signify the completion of a clock pulse). In the latter case, the zero bounds are true because in Algorithm *Ser* there can only be a message in transit if the end task is waiting for its node to become a sink, and cannot therefore be sending any messages (as these would be edge-reversal messages). For many algorithms, however, such bounds are either unknown, or known imprecisely, or simply do not exist. In such cases, the value of $r(c)$ should be set to a "large" positive integer M for all channels $c \in E_T$ whose bounds cannot be determined precisely. Just how large this M has to be, and what the limitations of this approach are, we discuss later in this section.

If the value of $r(c)$ is known precisely for all $c \in E_T$, then obviously the strategy of assigning $b(c) = r(c)$ buffers to every channel c guarantees the introduction of no additional deadlock, as every message ever to be in transit when its destination is engaged in a message transmission will be buffered (there may be more messages in transit, but only when their destination is not engaged in a message transmission, and will therefore be ready for reception within a finite amount of time). The interesting question here is, however, whether it can still be guaranteed that no new deadlock will be introduced if $b(c) < r(c)$ for some channels $c \in E_T$. This would be an important strategy to deal with the cases in which $r(c) = M$ for some $c \in E_T$, and to allow (potentially) substantial space savings in the process of buffer assignment. Theorem A.1 given next concerns this issue.

Theorem A.1. *Suppose that the distributed algorithm given by Algorithm Task(t) for all $t \in N_T$ is deadlock-free. Suppose in addition that G_T contains no directed cycle whose channels c are all such that $b(c) < r(c)$ or $r(c) = M$. Then the distributed algorithm obtained by replacing each infinite-capacity channel $c \in E_T$ with a $b(c)$-capacity channel is deadlock-free.*

Proof: A necessary condition for a deadlock to arise is that a directed cycle exists in G_T whose tasks are all blocked on an attempt to send messages on the channels on that cycle. By the hypotheses, however, every directed cycle in G_T has at least one channel c for which $b(c) = r(c) < M$, so at least the tasks t that have such channels in $Out(t)$ are never indefinitely blocked upon attempting to send messages on them. ■

The converse of Theorem A.1 is also true in general, but not always. Specifically, there may be cases in which $r(c) = M$ for all the channels c of a directed cycle, and yet the resulting algorithm is deadlock-free, as M may be a true upper bound for c (albeit unknown). So setting $b(c) = r(c)$ for this channel does not necessarily mean providing it with insufficient buffering space.

As long as we comply with the sufficient condition given by Theorem A.1, it is then possible to assign to some channels $c \in E_T$ fewer buffers than $r(c)$ and still guarantee that the resulting distributed algorithm is deadlock-free if it was deadlock-free to begin with. In the remainder of this section, we discuss two criteria whereby these channels may be selected. Both criteria lead to intractable optimization problems (i.e., *NP*-hard problems), so heuristics need to be devised to approximate solutions to them (some are provided in the literature).

The first criterion attempts to save as much buffering space as possible. It is called the ***space-optimal criterion***, and is based on a choice of M such that

$$M > \sum_{c \in E_T - C^+} r(c),$$

where C^+ is the set of channels for which a precise upper bound is not known. This criterion requires a subset of channels $C \subseteq E_T$ to be determined such that every directed cycle in G_T has at least one channel in C, and such that

$$\sum_{c \in C} r(c)$$

is minimum over all such subsets (clearly, C and C^+ are then disjoint, given the value of M, unless C^+ contains the channels of an entire directed cycle from G_T). Then the strategy is to set

$$b(c) = \begin{cases} r(c), & \text{if } c \in C; \\ 0, & \text{otherwise,} \end{cases}$$

which ensures that at least one channel c from every directed cycle in G_T is assigned $b(c) = r(c)$ buffers (Figure A.1). By Theorem A.1, this strategy then produces a deadlock-free result if no directed cycle in G_T has all of its channels in the set C^+. That this strategy employs the minimum number of buffers comes from the optimal determination of the set C.

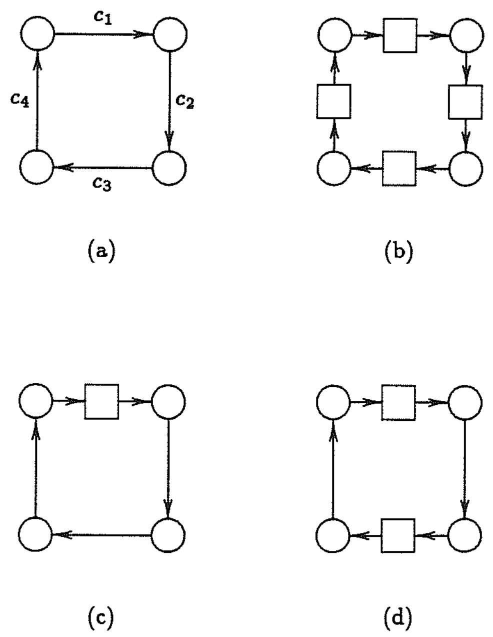


Figure A.1. *A graph G_T is shown in part (a). In the graphs of parts (b) through (d), circular nodes are the nodes of G_T, while square nodes represent buffers assigned to the corresponding channel in G_T. If $r(c) = 1$ for all $c \in \{c_1, c_2, c_3, c_4\}$, then parts (b) through (d) represent three distinct buffer assignments, all of which deadlock-free. Part (b) shows the obvious strategy of setting $b(c) = r(c)$ for all $c \in \{c_1, c_2, c_3, c_4\}$. Parts (c) and (d) represent, respectively, the results of the space-optimal and the concurrency-optimal strategies.*

The space-optimal approach to buffer assignment has the drawback that the concurrency in inter-task communication may be too low, inasmuch as many channels in E_T may be allocated zero buffers. Extreme situations can happen, as for example the assignment of zero buffers to all the channels of a long directed path in G_T. A scenario might then happen in which all tasks in this path (except the last one) would be suspended to communicate with its successor on the path, and this would only take place for one pair of tasks at a time. When at least one channel c has insufficient buffers (i.e., $b(c) < r(c)$) or is such that $r(c) = M$, a measure of concurrency that attempts to capture the effect we just described is to take the minimum, over all directed paths in G_T whose channels c all have $b(c) < r(c)$ or $r(c) = M$, of the ratio

$$\frac{1}{L+1},$$

where L is the number of channels on the path. Clearly, this measure can be no less than $1/|N_T|$ and no more than $1/2$, as long as the assignment of buffers conforms to the hypotheses of Theorem A.1. The value of $1/2$, in particular, can only be achieved if no directed path with more than one channel exists comprising channels c such that $b(c) < r(c)$ or $r(c) = M$ only.

Another criterion for buffer assignment to channels is then the *concurrency-optimal criterion*, which also seeks to save in buffering space, but not to the point that the concurrency as we defined might be compromised. This criterion looks for buffer assignments that yield a level of concurrency equal to $1/2$, and for this reason does not allow any directed path with more than one channel to have all of its channels assigned insufficient buffers. This alone is, however, insufficient for the value of $1/2$ to be attained, as for such it is also necessary that no directed path with more than one channel contain channels c with $r(c) = M$ only. Like the space-optimal criterion, the concurrency-optimal criterion utilizes a value of M such that

$$M > \sum_{c \in E_T - C^+} r(c).$$

This criterion requires a subset of channels $C \subseteq E_T$ to be found such that no directed path with more than one channel exists in G_T comprising channels from C only, and such that

$$\sum_{c \in C} r(c)$$

is maximum over all such subsets (clearly, $C^+ \subseteq C$, given the value of M, unless C^+ contains the channels of an entire directed path from G_T with more than one channel). The strategy is then to set

$$b(c) = \begin{cases} 0, & \text{if } c \in C; \\ r(c), & \text{otherwise,} \end{cases}$$

thereby ensuring that at least one channel c in every directed path with more than one channel in G_T is assigned $b(c) = r(c)$ buffers, and that, as a consequence, at least one channel c from every directed cycle in G_T is assigned $b(c) = r(c)$ buffers as well (Figure A.1). By Theorem A.1, this strategy then produces a deadlock-free result if no directed cycle in G_T has all of its channels in the set C^+. The strategy also provides concurrency equal to $1/2$ by our definition, as long as C^+ does not contain all the channels of any directed path in G_T with more than one channel. Given this constraint that optimal concurrency must be achieved (if possible), then the strategy employs the minimum number of buffers, as the set C is optimally determined.

A.3. TASK ALLOCATION

Once buffers have been assigned to the channels in E_T, the next step in the development of a distributed parallel program is to choose a processor for each task to run on. This is the *task allocation* step, which employs an additional graph, the undirected graph $G_P = (N_P, E_P)$, in the assignment of tasks to processors. This graph represents the actual distributed parallel machine on which the program is to be executed. Each member of N_P is then a *processor*, and each member of E_P is a bidirectional *communication link*.

Allocating tasks to processors requires that we find an *allocation function*, i.e., a mapping

$$A : N_T \to N_P$$

such that $A(t) = p$ if and only if task $t \in N_T$ is to run on processor $p \in N_P$. Here we should make the important observation that, although in this section we describe the task allocation process in terms of the graph G_T, this graph need not necessarily be the same as the one used in Section A.2, as G_T may now contain additional tasks to account for the message buffering activities discussed in Section A.2. Another observation is that the graph G_T need not necessarily reflect the actual arrangement in terms of tasks of the final distributed parallel program. After an allocation function A has been determined, it is customary, for instance, to create one single task per processor when coding the final program. The task allocation step has then in these cases the meaning of a "data partitioning" step. We return to this point in Section B.3 when we describe Occam programs for automaton network simulation.

Associated with both G_P and G_T are attributes that aid in the determination of a good allocation function. We assume first of all that G_P has a fixed *routing structure*, i.e., a structure determining the fixed route to be followed by messages sent between processors in G_P that are not directly connected by a communication link. Such a structure is given by the mapping

$$R : N_P \times N_P \to 2^{E_P}$$

such that $R(p,q)$ is the set of links on the route from processor p to processor q, possibly distinct from $R(q,p)$, and such that $R(p,p) = \emptyset$, for all $p,q \in N_P$. Additional attributes of G_P are the relative *processor speed* (in instructions per unit time) of $p \in N_P$, s_p, and the relative *link capacity* (in bits per unit time) of $(p,q) \in E_P$, $c_{(p,q)}$ (the same in both directions). These numbers are such that the ratio s_p/s_q indicates how faster processor $p \in N_P$ is than processor $q \in N_P$; similarly for the communication links.

The attributes of graph G_T are the following. Each task $t \in E_T$ is represented by a relative *processing demand* (in number of instructions) ψ_t, while each channel $(t,u) \in E_T$ is represented by a relative *communication demand* (in number of bits) from task t to task u, $\zeta_{(t,u)}$, possibly different from $\zeta_{(u,t)}$. The ratio ψ_t/ψ_u is again indicative of how much more processing task $t \in N_T$ requires than task $u \in N_T$, the same holding for the communication requirements.

The process of task allocation is generally viewed as one of two main possibilities. It may be *static*, if the allocation function A is determined prior to the beginning of the distributed parallel computation and kept unchanged for its entire duration, or it may be *dynamic*, if A is allowed to change during the course of the distributed parallel computation. The former approach is suitable to cases in which both G_P and G_T, as well as their attributes, vary negligibly with time. The dynamic approach, on the other hand, is more appropriate to cases in which either the graphs or their attributes are time-varying, and then provides opportunities for the allocation function to be revised in the light of such changes. What we describe in the remainder of this section are approaches to the static allocation of tasks to processors. The dynamic case is usually much more difficult, as it requires tasks to be migrated among processors, thereby interfering with the ongoing computation. Successful results of such dynamic approaches are for this reason scarce, except for some attempts that can in fact be regarded as a periodic repetition of the calculations for a static task allocation, whose resulting allocation functions are then kept unchanged for the duration of the period.

The quality of an allocation function A is normally measured by a function that expresses the time for completion of the entire computation, or some function of this time. This criterion is not accepted as a consensus, but it seems to be consonant with the overall goal of parallel processing systems, namely to compute faster. So obtaining an allocation function by the minimization of such a function is what one should seek. The function we utilize in this book to evaluate the efficacy of an allocation function A is the function $H(A)$ given by

$$H(A) = \alpha H_P(A) + (1-\alpha) H_C(A),$$

where $H_P(A)$ gives the time spent with computation when A is followed, $H_C(A)$ gives the time spent with communication when A is followed, and α such that $0 < \alpha < 1$ regulates the relative importance of $H_P(A)$ and $H_C(A)$. This parameter α is crucial, for example, in conveying to the task allocation process some information on how efficient the routing mechanisms for interprocessor communication are (we discuss some of these in Section A.4).

These two components of $H(A)$ are given respectively by

$$H_P(A) = \sum_{p \in N_P} \left(\sum_{t \in N_T \mid A(t)=p} \frac{\psi_t}{s_p} + \sum_{t,u \in N_T \mid t \neq u, A(t)=A(u)=p} \frac{\psi_t \psi_u}{s_p} \right)$$

and

$$H_C(A) = \sum_{(p,q) \in E_P} \frac{1}{c_{(p,q)}} \sum_{(t,u) \in E_T \mid (p,q) \in R(A(t),A(u))} \zeta_{(t,u)}.$$

This definition of $H_P(A)$ has two types of components. One of them, ψ_t/s_p, accounts for the time to execute task $t \in N_T$ on processor $p \in N_P$. The other component, $\psi_t\psi_u/s_p$, is a function of the additional time incurred by processor $p \in N_P$ when executing both tasks $t, u \in N_T$ (various other functions can be used here, as long as nonnegative). If an allocation function A is sought by simply minimizing $H_P(A)$, then the first component will tend to lead to an allocation of tasks to the fastest processors, while the second component will lead to a dispersion of the tasks among the processors. The definition of $H_C(A)$, in turn, embodies components of the type $\zeta_{(t,u)}/c_{(p,q)}$, which reflects the time spent in communication from task $t \in N_T$ to task $u \in N_T$ on link $(p,q) \in R(A(t), A(u))$. Contrasting with $H_P(A)$, if an allocation function A is sought by simply minimizing $H_C(A)$, then tasks will tend to be concentrated on a few processors. The minimization of the overall $H(A)$ is then an attempt to reconcile conflicting goals, as each of its two components tend to favor different aspects of the final allocation function.

As an example, consider the two-processor system comprising processors p and q. Consider also the two tasks t and u. If the task allocation function A_1 assigns t to run on p and u to run on q, then we have, assuming $\alpha = 1/2$,

$$2H(A_1) = \frac{\psi_t}{s_p} + \frac{\psi_u}{s_q} + \frac{\zeta_{(t,u)} + \zeta_{(u,t)}}{c_{(p,q)}}.$$

A task allocation function A_2 assigning both t and u to run on p yields

$$2H(A_2) = \frac{\psi_t}{s_p} + \frac{\psi_u}{s_p} + \frac{\psi_t\psi_u}{s_p}.$$

Clearly, the choice between A_1 and A_2 depends on how the system's parameters relate to one another (Figure A.2). For example, if $s_p = s_q$, then A_1 is preferable if the additional cost of processing the two tasks on p is higher than the cost of communication between them over the link (p,q), that is, if

$$\frac{\psi_t\psi_u}{s_p} > \frac{\zeta_{(t,u)} + \zeta_{(u,t)}}{c_{(p,q)}}.$$

Finding an allocation function A that minimizes $H(A)$ is a very difficult problem, *NP*-hard in fact, as so many others we have encountered in this book. It remains *NP*-hard even if we restrict the possibilities of the processor graph N_P to the class of hypercube machines, today of great importance. Given this inherent difficulty, all that is left is to resort to heuristics that allow a "satisfactory" allocation function to be found, that is, an allocation function that can be found reasonably fast and that does not lead to a poor performance of the distributed

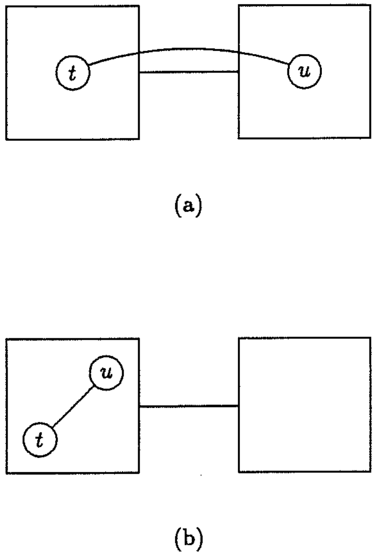


Figure A.2. *In this figure, square nodes represent processors and circular nodes represent tasks. Parts (a) and (b) depict two different allocations of tasks to processors. Depending on how the overhead of executing both t and u on a same processor (b) compares to the cost of communication over the link between the processors (a), one allocation may be preferable or the other.*

parallel program. We provide in the remainder of this section a brief overview of two such heuristics.

Our first heuristic is the A^* algorithm, which is a heuristic state-space search method. In the case of our task allocation problem, each *state* is a *partial allocation* of tasks to processors, i.e., an allocation function that may leave tasks without processors to run on. Starting at a state n_0 corresponding to a situation in which no task is allocated, A^* generates a sequence of states $n_0, \ldots, n_e$ such that n_e corresponds to a complete allocation. When in state n_k for $0 \le k < e$, A^* generates n_{k+1} by following the steps that we describe next, where *Set* is a data structure used to maintain a set of states and $f(n_k)$ is the "cost" of state n_k, to be discussed in more detail shortly.

1. $Set := \emptyset$;
2. $k := 0$;
3. Let t be a task that is not allocated in n_k. If none can be found, then proceed to step 6;
4. Generate one new state n for each $p \in N_P$ by allocating t to p. Add n to *Set*;
5. Let $n_{k+1} \in Set$ be such that $f(n_{k+1}) \le f(n)$ for all $n \in Set$. Remove n_{k+1} from *Set*, increment k by 1, and go to step 3;
6. $e := k$.

For $0 \le k \le e$, let $N_k \subseteq N_T$ be the set of tasks allocated in state n_k, and let $A_k : N_k \to N_P$ be the corresponding partial allocation function. The measure $f(n_k)$ used by A^* is given by the sum

$$f(n_k) = g(n_k) + h(n_k),$$

where $g(n_k) = H(A_k)$ (this is given by $H(A)$ by considering the tasks in N_k only) and $h(n_k)$, the ***heuristic function***, is an estimate of the difference $H(A_e) - H(A_k)$, i.e., of the "cost" to complete the partial allocation in n_k to yield the complete allocation in n_e. If we let $h^*(n_k)$ denote the optimal "cost" of obtaining a solution from n_k, then it is known from the literature that n_e is an optimal state (i.e., one of globally minimum $H(A)$) if $h(n_k) \le h^*(n_k)$ for all $0 \le k \le e$. The heuristic function is then said to be ***admissible***.

One admissible heuristic function is

$$h(n_k) = \sum_{t \in N_T - N_k} \min_{p \in N_P} \left\{ \frac{\psi_t}{s_p} + \sum_{u \in N_k \mid A_k(u) = p} \frac{\psi_t \psi_u}{s_p} \right.$$
$$\left. + \sum_{u \in N_k} \left(\sum_{(q,r) \in R(p, A_k(u))} \frac{\zeta_{(t,u)}}{c_{(q,r)}} + \sum_{(q,r) \in R(A_k(u), p)} \frac{\zeta_{(u,t)}}{c_{(q,r)}} \right) \right\},$$

where the communication among tasks in $N_T - N_k$ is not taken into account, nor is the overhead of processing two tasks in $N_T - N_k$ on a same processor (thence the admissibility).

Various improvements can be performed on this application of the A^* search technique to the task allocation problem, aiming at improved efficiency. Things like limiting the maximum size of *Set* and imposing an upper bound on the distance between any two processors to which communicating tasks may be allocated are usually effective. In addition, there are specialized data structures for the implementation of *Set* that allow the operations of insertion and removal of states in nondecreasing order of f to be done efficiently. Finally, it is in general advantageous to try to select tasks for allocation in step 3 by starting with those that have large processing and communication demands. This tends to yield an h that is closer to the real value of the remaining "cost," thereby leading the search, as is known from the literature, to resemble something like a depth-first search more than a breadth-first search, therefore faster. With this purpose, tasks t may then be kept on a pre-sorted list in nonincreasing value of

$$\psi_t + \sum_{u \in N_T | u \neq t} (\zeta_{(t,u)} + \zeta_{(u,t)}).$$

The second heuristic we discuss in this appendix for finding a task allocation function A is based on the Hopfield-Tank model for combinatorial optimization we described in Section 6.3, and utilizes an analog Hopfield neural network. The problem we used as example in that section was TSP. The approach here is in fact quite similar to the one employed in the solution of TSP, so we describe it in much less detail than we did in Section 6.3 for TSP. In this section, our notation for the real-time parameter will differ slightly from Chapters 6 and 7: we shall denote it by τ, as the usual t already denotes a member of N_T.

The analog Hopfield neural network we employ has $|N_T||N_P|$ neurons arranged as an $|N_T| \times |N_P|$ two-dimensional array (Figure A.3). For $t \in N_T$ and $p \in N_P$, the neuron occupying the position on row t and column p in the array is denoted by n_{tp}. An allocation function A is represented at time $\tau \geq 0$ in this array of neurons by an arrangement of neuron states such that

$$v_{tp}(\tau) \approx \begin{cases} 1, & \text{if } A(t) = p; \\ 0, & \text{otherwise} \end{cases}$$

for all $t \in N_T$ and all $p \in N_P$. Surely, this means that for a task $t \in N_T$ there has to be exactly one processor $p \in N_P$ such that $v_{tp}(\tau) \approx 1$, as t has to be allocated to exactly one processor by A. This corresponds in the array of neurons to a situation in which exactly $|N_T|$ neurons have their states close to 1, of which there is exactly one per row. The remaining neurons have their states close to 0.

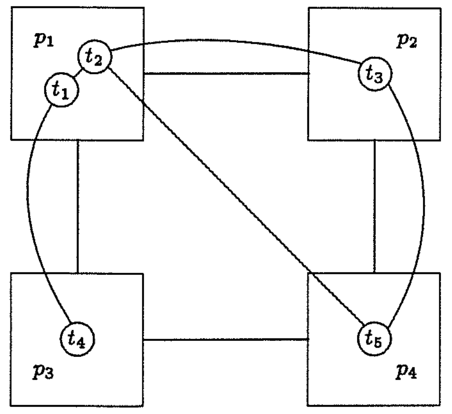
Similarly to what we did in Section 6.3, these "syntactic" constraints are enforced as follows. First, neurons with high-gain (high values of γ) sigmoid transfer functions are employed, so that by Theorem 6.2 we know that the network always stabilizes with all neurons' states close to either 0 or 1. Secondly, every two neurons on the same row are interconnected by a synaptic strength $W < 0$ such that $|W|$ is "high enough" to inhibit the situations in which both neurons have states close to 1. Finally, each neuron's threshold potential is set to a value negative enough for at least one neuron per row to be in a state close to 1 when the network stabilizes.

The remaining parameters of the neural network are set as follows. For $t \in N_T$ and $p \in N_P$, the external input to neuron n_{tp} is

$$e_{tp} = -\frac{\psi_t}{s_p}.$$

For $t, u \in N_T$ and $p \in N_P$, the synaptic strength between neurons n_{tp} and n_{up} is

$$w_{tp,up} = -\frac{\psi_t \psi_u}{s_p}.$$



(a)

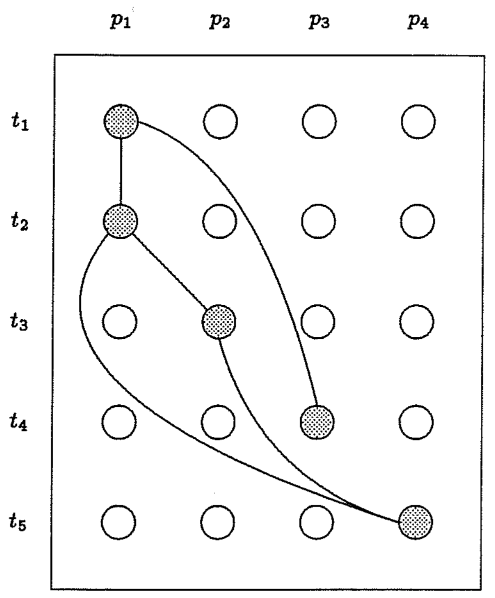
Figure A.3. *In part (a), we show an allocation of tasks $t_1, \ldots, t_5$ (shown as circular nodes) to processors $p_1, \ldots, p_4$ (shown as square nodes). In part (b), the $|N_T| \times |N_P|$ (in this case, 5×4) array of neurons to solve the task allocation problem is shown in a stable state corresponding to the allocation of part (a). Shaded neurons are in states close to 1, and the others are in states close to 0. Only the connections between neurons with states close to 1 are shown.*

Finally, for $t, u \in N_T$ and $p, q \in N_P$, the synaptic strength between neurons n_{tp} and n_{uq} is

$$w_{tp,uq} = -\sum_{(r,s) \in R(p,q)} \frac{\zeta_{(t,u)}}{c_{(r,s)}} - \sum_{(r,s) \in R(q,p)} \frac{\zeta_{(u,t)}}{c_{(r,s)}}$$

(Figure A.3).

Let θ and R be, respectively, the threshold potential and the resistance of all neurons. Theorem A.2 is the counterpart of Theorem 6.3. It gives sufficient conditions for the analog Hopfield neural network we just described to be in a feasible state whenever it stabilizes. A *feasible state* here is a state that represents a task allocation function.



(b)

Figure A.3 (continued)

Theorem A.2. *If*

$$W < \frac{\theta}{R} < \min_{\substack{t\in N_T\\ p\in N_P}} \left\{ -\frac{\psi_t}{s_p} - \sum_{u\in N_T|u\neq t}\left(\frac{\psi_t\psi_u}{s_p} + \sum_{q\in N_P|q\neq p}\left(\sum_{(r,s)\in R(p,q)}\frac{\zeta_{(t,u)}}{c_{(r,s)}} + \sum_{(r,s)\in R(q,p)}\frac{\zeta_{(u,t)}}{c_{(r,s)}}\right)\right)\right\},$$

then every stable state of the neural network is feasible.

Proof: Our proof here is entirely analogous to the proof of Theorem 6.3. That is, we suppose, to the contrary of the theorem's assertion, that for some time $\tau \geq 0$ a stable state of the neural network is infeasible. Then one of the following three cases must happen.

(i) There are $|N_T|$ neurons with states close to 1, but not one per row.

(ii) There are more than $|N_T|$ neurons with states close to 1.

(iii) There are less than $|N_T|$ neurons with states close to 1.

If case (i) or case (ii) holds, then at least one row (say t) has $x > 1$ neurons with states close to 1. If p is the column of one such neuron, then $v_{tp}(\tau) \approx 1$. In this case, $u_{tp}(\tau) > \theta$, and then (6.2) yields

$$C_{tp}\frac{du_{tp}(\tau)}{d\tau} < (x-1)W - \frac{\theta}{R},$$

where we have taken into account the strict negativity of the external input to n_{tp} and of the synaptic strengths that connect n_{tp} outside its row or column. By the first inequality in the hypothesis, and considering that $x > 1$, it then follows that

$$\frac{du_{tp}(\tau)}{d\tau} < 0.$$

Similarly, if case (iii) holds, then there must exist one row (say t) whose neurons all have states close to 0. In this case, letting p be any column, $u_{tp}(\tau) < \theta$, and then, by (6.2),

$$C_{tp}\frac{du_{tp}(\tau)}{d\tau} > -\frac{\psi_t}{s_p} - \sum_{u\in N_T|u\neq t}\left(\frac{\psi_t\psi_u}{s_p} + \sum_{q\in N_P|q\neq p}\left(\sum_{(r,s)\in R(p,q)}\frac{\zeta_{(t,u)}}{c_{(r,s)}} + \sum_{(r,s)\in R(q,p)}\frac{\zeta_{(u,t)}}{c_{(r,s)}}\right)\right) - \frac{\theta}{R}.$$

By the second inequality in the hypothesis, we then have

$$\frac{du_{tp}(\tau)}{d\tau} > 0.$$

In conclusion, we have a neuron n_{tp} for which either $u_{tp}(\tau) > \theta$ and $du_{tp}(\tau)/d\tau < 0$ or $u_{tp}(\tau) < \theta$ and $du_{tp}(\tau)/d\tau > 0$. The neural network state must then be unstable. ■

By (6.4), it follows from Theorem A.2 that the neural network's energy is, at stable states, given by

$$H(A) + |N_T|\frac{\theta}{R}$$

for some allocation function A. It also follows from Theorem A.2 that the optimal allocation function corresponds to a stable state of the neural network, so the problem of optimally allocating tasks to processors is equivalent to minimizing the network's energy, for example by means of the algorithms discussed in Section 6.2.

A.4. INTERPROCESSOR COMMUNICATION

In Section A.3, we have discussed the allocation of tasks to processors, and have seen that it is a possibility that two neighbor tasks in G_T be allocated to distinct processors or even to processors that are not neighbors in G_P. A message sent from task $t \in N_T$ to task $u \in N_T$ then has to be sent on all the links in $R(p,q)$, the set of links on the route from processor $p \in N_P$ to processor $q \in N_P$, assuming that t has been allocated to p and u to q. If the routes given by R are shortest paths in G_P, then at each processor $r \in N_P$ adjacent to a link in $R(p,q)$, $next_r(q)$ indicates which of the links in $R(p,q)$ is to be used as the first hop toward q. This indication is sufficient to represent R, as long as R has been built such that $R(r,q) \subseteq R(p,q)$, which is certainly a possibility. The function *next* is a *routing function.*

In most distributed parallel processing systems, each processor in G_P is equipped with a special module called the *communication processor.* This communication processor receives all messages that outgo from tasks running on that processor and sends them on to the processors on which the destination tasks reside. In addition, the communication processor is also responsible for forwarding to the locally residing tasks whichever messages it receives from other processors destined to those tasks. A message *msg* sent to a task u residing on processor q is received by the origin communication processor as simply (u, msg), and treated thereafter as the more complex message (q, u, msg), where the prefix q is added by the communication processor to indicate the destination processor (this information is given by the allocation function — cf. Section A.3 — and is replicated throughout all processors). Upon receiving a message (q, u, msg), the communication processor at a processor $p \in N_P$ either retains *msg* so that it can be forwarded to task t, if $q = p$, or sends it on link $next_p(q)$, otherwise.

There are many variations on this general scheme for the transport of messages between processors. Most of these variations are accounted for by differences in the way messages incoming to a communication processor are treated. We briefly review in this section three mechanisms for the routing of messages in distributed parallel systems. They constitute notable steps in the evolution of interprocessor communication mechanisms, aiming at reducing the time for messages to be transported between processors. Several of the concepts involved are inherited from the research on wide-area data networks in previous decades, and can be applied directly.

The first mechanism for routing is known as *store-and-forward.* It requires a message (q, u, msg) to be broken into *packets* of fixed size. Each packet carries the same addressing information as the original message (q and u), and can therefore be transmitted independently. If these packets cannot be guaranteed to be delivered in the order they are sent, then they must also carry a sequence number, to be used at the destination processor for re-assembly of the message. At intermediate processors, packets are stored in buffers for later transmission when the required outgoing link becomes available (a queue of packets is kept for each outgoing link).

Store-and-forward routing is prone to the occurrence of deadlocks, as the packets compete for shared resources (buffering space, in this case). One simple situation in which this may happen is the following. Consider a cycle of processors in G_P, and

suppose that one task in each of these processors has a packet to send to another task residing at another processor on the cycle that is more than one hop away. Suppose further that the routing function given by *next* is such that all these tasks attempt to send their packets in the same direction (clockwise or counter-clockwise) on the cycle of processors. If buffering space is no longer available at any of the processors on the cycle, then deadlock is certain to occur.

The usual strategy to prevent this type of deadlock is to employ a *structured buffer pool.* This is a mechanism whereby the buffers at all processors are divided into classes. Whenever a packet is sent between two neighbor processors, it can only be accepted for storage at the destination processor if there is buffering space in a specific buffer class, which is normally a function of some of the packet's end-to-end route's parameters. If this function allows no cyclic dependency to be formed among the various buffer classes, then deadlock is ensured never to occur. Some routing functions are themselves almost acyclic, and as a consequence render the task of establishing buffer classes rather simple. The *e-cube routing function* for hypercubes, for example, is such that only two buffer classes are needed at a processor, one for packets incoming from the local tasks, the other for packets that arrive from neighbor processors.

The store-and-forward routing mechanism suffers from two main drawbacks: the latency for end-to-end delivery of the message (as the packets have to be stored at all intermediate processors), and the need to use memory bandwidth, which seldom can be provided entirely by the communication processor and has then to be shared with the locally running tasks.

The potentially excessive latency of store-and-forward routing is partially remedied by the second mechanism we describe, *circuit switching.* This mechanism requires an end-to-end route to be entirely reserved for a message before it is transmitted. Once all the links on the route are secured for that particular transmission, the message is then sent and at the intermediate processors incurs no additional delay waiting for links to become available. The reservation process employed by circuit switching is also prone to the occurrence of deadlocks, as links may participate in several routes, portions of which may form directed cycles that may deadlock the reservation of links. The e-cube routing function for hypercubes is once more especially suitable, as the only directed cycles it allows necessarily involve full end-to-end routes, so the reservation of intermediate links is guaranteed to progress without any deadlocks.

Circuit switching is obviously inefficient for the transmission of short messages, as the time for the entire path to be reserved becomes then prominent. Even for long messages, however, its advantages may not be too pronounced, depending primarily on how the message is transmitted once the links are reserved. If the message is broken into packets that have to be stored at the intermediate processors, then the gain with circuit switching may be only marginal, as the packet is only sent on the next link after it has been completely received on the previous one (all that is saved is then the wait time on outgoing packet queues). It is possible, however, to pipeline the transmission of the message so that only very small portions have to be stored at the intermediate processors, as in the third routing strategy we now describe.

Buffering packets as we have described for the store-and-forward and the circuit switching routing mechanisms is an approach to *flow control*, which is in fact the basic issue we have to deal with when selecting a routing strategy. Essentially, flow control is a shared-resource management policy, determining rules for the allocation of links and buffering space. Besides packet buffering, another important flow control strategy employs packet blocking as one of its basic paradigms. The resulting routing strategy is known as *wormhole routing*. As opposed to the previous two strategies, the basic unit on which flow control is performed in wormhole routing is not a packet but a *flit* (*flow-control digit*). A flit contains no routing information, so every flit in a packet must follow the leading flit, where the routing information was kept when the packet was subdivided.

With wormhole routing, the inherent latency of store-and-forward routing due to the constraint that a packet can only be sent forward after it has been received in its entirety is eliminated. All that needs to be stored is a flit, significantly smaller than a packet, so the transmission of the packet is pipelined, as portions of it may be flowing on different links and portions may be stored. When a flit needs access to a resource (memory space or link) that it cannot have immediately, the entire packet is blocked and only proceeds when the leading flit can advance. If the routing function is not acyclic, then as in the previous two mechanisms deadlock can arise. The strategy for dealing with this in wormhole routing is to break the cycles in the routing function (thereby possibly making pairs of processors inaccessible to each other), then add *virtual links* to the already existing links in the network, and then finally fix the routing function by the use of the virtual links. Cycles in the routing function then become "spirals," and deadlocks can no longer occur. (Virtual links are in the literature referred to as *virtual channels*, but channels have had in this appendix a different connotation.)

To finalize this section, we mention that yet another routing strategy, which can be regarded as a hybrid strategy combining store-and-forward and wormhole routing, is called *virtual cut-through*. In this strategy, the transmission of packets is pipelined as in wormhole routing, but entire packets are stored when an outgoing link cannot be immediately used, as in store-and-forward. Virtual cut-through can then be regarded as a variation of wormhole routing in which the pipelining in packet transmission is retained but packet blocking is replaced with packet buffering.

A.5. BIBLIOGRAPHIC NOTES

General references for the material in Section A.1 are Ben-Ari (1982), Andrews and Schneider (1983), Peterson and Silberschatz (1985), Maekawa, Oldehoeft, and Oldehoeft (1987), Perrott (1987), and Andrews (1991). Guarded commands, in particular, were introduced in Dijkstra (1975), while CSP is discussed by Hoare (1978, 1984). The issue of blocking versus nonblocking communication primitives has been studied somewhat extensively. Although there are arguments supporting the intuitive expectation that "more concurrency" can be obtained with the use of nonblocking communication (Gentleman, 1981), it has been shown that such is not necessarily the case (Barbosa, 1990b). This result is based on the concept of edge

multicolorings in graphs, treated by Fiorini and Wilson (1977) and Stahl (1979), and how they relate to special polyhedra that appear in connection with matchings in graphs (Edmonds, 1965; Fulkerson, 1972).

The material on buffer allocation in Section A.2 is based on Barbosa (1990a), where a heuristic for the concurrency-optimal buffer allocation is also presented. This heuristic is based on the computation of a maximum weighted matching on a graph, for which an efficient algorithm is described in Syslo, Deo, and Kowalik (1983).

References on the allocation of tasks to processors are Ma, Lee, and Tsuchiya (1982), Shen and Tsai (1985), Fox, Kolawa, and Williams (1987), Sinclair (1987), and Nicol and Reynolds (1990). The intractability of optimal task allocation on hypercubes is shown by Krumme, Venkataraman, and Cybenko (1986). The task allocation model presented in Section A.3 is from Barbosa and Huang (1988). References on the A^* algorithm are Nilsson (1980) and Pearl (1984), while data structures for use with it can be found among the ones given in Cormen, Leiserson, and Rivest (1990). Freitas and Barbosa (1991) present results of an experimental evaluation of distributed parallel variants of this algorithm. Another approach that utilizes neural networks for the allocation of tasks has been given by Fox and Furmanski (1988).

The issue of interprocessor communication in distributed parallel machines has benefited greatly from the research on computer networks of previous decades. Bertsekas and Gallager (1987) can be looked up for the aspects of interest within the context of Section A.4. Communication processors are discussed in Dally, Chao, Chien, Hassoun, Horwat, Kaplan, Song, Totty, and Wills (1987), Ramachandran, Solomon, and Vernon (1987), and Dally (1990), where the *e*-cube routing function is also described. For information concerning some commercially available machines, and how the problems of interprocessor communication were solved in their designs, the reader should refer to Arlauskas (1988), Grunwald and Reed (1988), and Pase and Larrabee (1988). A survey of the various issues involved in flow control is given by Gerla and Kleinrock (1982), where the traditional approaches to deadlock prevention in message routing are treated (another deadlock-prevention strategy in this context is given by Günther (1981)). Kermani and Kleinrock (1979) describe virtual cut-through, and wormhole routing is found in Dally and Seitz (1987).

B

The software for automaton network simulation

The purpose of this appendix is to demonstrate how the distributed parallel algorithms given throughout the book for automaton network simulation can be coded in an existing programming language. The language we use for illustration is Occam, which appears to be very well suited for programming parallel applications of this type. A brief review of Occam is given in Section B.1, and then in Section B.2 we discuss the main points of a generic procedure to obtain Occam programs from distributed algorithms. The outline of an Occam program for automaton network simulation is given in Section B.3. Bibliographic notes follow in Section B.4.

B.1. OCCAM PROGRAMMING

In this section, we present a very brief overview of the main characteristics of the Occam programming language. As will become apparent as we progress, Occam is very suitable to the programming of distributed parallel algorithms with the general appearance of Algorithm *Task*(t) presented in Section A.1, which can be regarded as a generic template for all the distributed parallel algorithms considered in this book. Although current versions of Occam lack very important facilities for data structure manipulation, the ease with which one can develop distributed parallel programs in Occam by far outweighs such shortcomings. Occam is inspired in the CSP (Communicating Sequential Processes) notation.

The main building blocks for actions in Occam programs are called *processes*, and can be of various types. A *primitive process* can be an *assignment process*, as for example

```
x:=y
```

which assigns variable y to variable x.

Constants and variables are declared in much the same way it is done in most modern programming languages. One variable type, however, is of significant importance within the context of distributed parallel programming. This is the `CHAN` type, used to declare communication channels for the communication between concurrently executing processes, which cannot in Occam access shared variables (except, of course, of the `CHAN` type) and therefore are constrained to communicating with one another solely by means of message passing on `CHAN` variables. A variable of type `CHAN` is a zero-capacity communication channel (cf. Sections A.1 and A.2), meaning that no buffering is provided and that the processes involved in the communication must synchronize with each other in order to resume their interaction.

Another primitive process is the *input process*, indicated as in

```
input.channel?x
```

where `input.channel` is a variable of type `CHAN` and `x` is another variable. This input process has the effect of assigning to variable `x` whatever is received on channel `input.channel`. Similarly, another primitive process is the *output process*, which is indicated as in

```
output.channel!y
```

In this output process, `output.channel` is a variable of type `CHAN` and `y` is another variable. The effect of this process is to send on channel `output.channel` the contents of `y`.

A primitive process is said to have *terminated* when the assignment is completed (for an assignment process), or the message has been received (for an input process), or the message has been sent (for an output process).

More elaborate types of processes are the *sequential process*, the *parallel process*, and the *alternative process*. A sequential process is built with the construct `SEQ`, as in

```
SEQ
  p1
  p2
```

where `p1` and `p2` are processes. Process `p2` is only executed after `p1` terminates. Upon termination of `p2`, the sequential process *terminates*.

A parallel process is built with the construct `PAR`, as for example in

```
PAR
  p1
  p2
```

where `p1` and `p2` are processes. Processes `p1` and `p2` may be executed concurrently, and the parallel process only *terminates* when both `p1` and `p2` have terminated. A parallel process can also be built with a `PRI PAR` construct, whose semantics is identical to that of `PAR`, except that processes are given nondecreasing execution priority in the order they appear inside the `PRI PAR`.

An alternative process implements the notion of guarded commands discussed in Section A.1. It is built with the aid of the construct `ALT`. For example, in

```
ALT
  g1
    p1
  g2
    p2
```

the alternative process comprises two guarded commands, whose guards are `g1` and `g2`, and whose commands are, respectively, the processes `p1` and `p2`. A guard is in Occam either an input process or the conjunction of a boolean condition with an input process, as in

```
(x>0)&input.channel?y
```

The presence of an input process in a guard has to be understood as a boolean condition as well, inasmuch as it is an indication that a message is ready for reception at the specified channel, in addition to the eventual reception itself. The functioning of an alternative process follows the semantics of a disjunction of guarded commands, as we discussed in Section A.1. Of those guards that are ready (i.e., both the boolean condition is true and there is a message for immediate reception), exactly one is selected (any one). The message is then actually received and the corresponding process is executed. After termination of this process, the alternative process *terminates*.

Sequential, parallel, and alternative processes may be replicated for a fixed number of times by means of a replicator denoted by `FOR`. If `n` is a constant and `i` is a variable, then, for example, a call to procedure `proc` can be replicated `n` times, each one parameterized by a value of `i` in the range 0 through $n-1$, as in

```
PAR i=0 FOR n
  proc(i)
```

The effect of this replicated `PAR` is equivalent to

```
PAR
  proc(0)
  ...
  proc(n-1)
```

Communication channels in Occam are, as we have seen, zero-capacity channels. From our discussion in Sections A.1 and A.2, it is clear that programming with such channels can be a difficult task, as the slightest error can lead to a deadlock when the program is run. What is often needed then is to "create" channels of higher capacity, so that the techniques discussed in Section A.2 for proper message buffering can be effectively employed. In the remainder of this section, we give two solutions to the problem of obtaining in Occam the effect of a positive-capacity channel between two parallel processes.

The first solution we describe employs the relay tasks described in Section A.2. Each relay task has a storage capacity of exactly one message, has one single input channel, and one single output channel. Whenever its buffer is empty, a relay task waits to receive another message on its input channel, buffers it, and then attempts to send it on its output channel (Figure B.1). An Occam version of this relay task which is allowed to loop forever might be written as follows.

```
WHILE TRUE
  SEQ
    input.channel?buffer
    output.channel!buffer
```

In this relay task, intput.channel and output.channel are of the CHAN type. A message incoming on input.channel is stored in buffer (the buffer), and then sent on output.channel. An arrangement of such relay tasks inside a PAR replicated to provide as many buffers as needed to account for the overall desired channel capacity can be used to construct positive-capacity communication channels. The only requirement is that the CHAN variables be used to arrange the relay tasks in tandem.

The same effect can be achieved with the use of much less parallelism, and consequently much less process scheduling effort by the transputer, which can be profitable in many situations. The other solution we discuss employs two tasks (for any desired overall capacity, instead of one task per buffer unit), one to hold one buffer unit, the other to hold the remaining buffer units. The two tasks are arranged in tandem in such a way that messages to be buffered are received by the task with the largest buffering capacity and stored in a bounded buffer. The other task is responsible for shipping messages out, and after it does so it requests another message from the larger buffering task. Note that this explicit backward request for an additional message is essential to ensure that the assembled tasks' buffers can be used to their full capacity (Figure B.1).

In the solution we give next, we employ four CHAN variables: input.channel for external inputs to the first task, output.channel for external outputs from the second task, and request.channel and grant.channel for communication between the two tasks. The set of buffers managed by the first task (the bounded buffer) is in our solution represented in a rather abstract way by the booleans buffer.not.full and buffer.not.empty, and by the buffer items first.empty.buffer and first.occupied.buffer. These should be easily obtainable from the usual implementation of a bounded buffer as a circular buffer. The buffer kept by the second task is buffer.

Our second suggestion for a positive-capacity channel follows. For any $k \geq 2$, this Occam fragment, with a bounded buffer of $k-1$ buffer units, implements a k-capacity channel (for $k = 1$, the previous solution should obviously be used; for $k = 2$, both solutions are acceptable, although the first one is more economical).

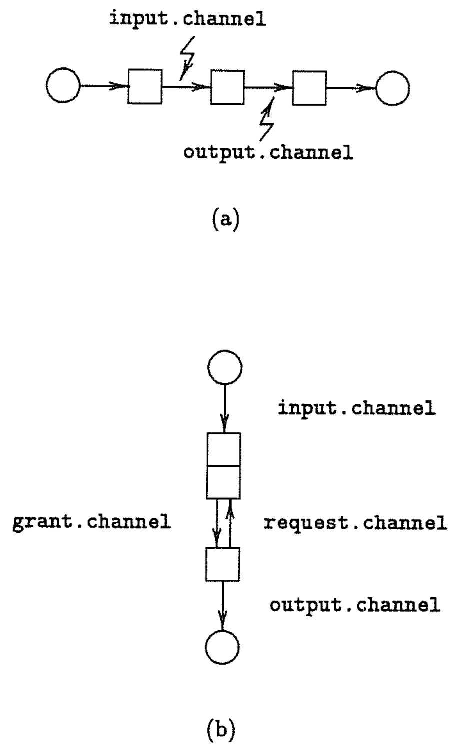


Figure B.1. *Two possibilities for the implementation in Occam of positive-capacity channels are shown in this figure. The two circular nodes in parts (a) and (b) represent the tasks between which the positive-capacity channel is needed. Each square represents a buffer for exactly one message. In part (a), positive capacity is achieved by employing relay tasks, one for each buffer needed. In part (b), useful when at least two buffers are needed, exactly two intermediate tasks are employed, one corresponding to the "head-of-the-line" buffer, the other corresponding to the remaining buffers. An additional backward request channel is employed to keep the flow of messages between the two units.*

```
PAR
  WHILE TRUE
    ALT
      (buffer.not.full)&input.channel?first.empty.buffer
        SKIP
      (buffer.not.empty)&request.channel?r1
        grant.channel!first.occupied.buffer
  WHILE TRUE
    SEQ
      request.channel!r2
      grant.channel?buffer
      output.channel!buffer
```

The SKIP command in the first guarded command is mandatory so that the command after the guard is not left unspecified. It would, in a real implementation, be substituted for by instructions to manipulate the bounded buffer (these would also be present in the second guarded command). Variables r1 and r2 are used solely for the exchange of a backward message-request signal between the two tasks.

B.2. OCCAM PROGRAMS FROM DISTRIBUTED ALGORITHMS

As we discussed throughout Appendix A, the process of turning a distributed parallel algorithm into a program that can be run on a real parallel machine involves several aspects, as the provision of buffers to accommodate the messages sent on the channels initially assumed to have infinite capacity, the allocation of tasks to processors, and the routing of messages among the various processors. Two important guidelines are the absence of deadlocks (unless the original algorithm is itself deadlock-prone) and overall efficiency. Many parallel processing systems already provide facilities for the proper routing of messages, so this issue does not have to be dealt with explicitly by the user. That is not the case, however, of transputer-based systems running Occam programs — an Occam program has to include modules to deal with every one of the aforementioned aspects, including buffering and routing.

Given the discussion in Appendix A and in Section B.1, we are now in a very comfortable position to design a generic Occam program embodying modules for every required function. Most of these modules can be thought of as belonging to a pre-developed Occam module library, and can then be employed by the user directly.

The template Occam program we describe shortly in this section includes various such modules, which we list next (see also Figure B.2). The graphs $G_T = (N_T, E_T)$ and $G_P = (N_P, E_P)$ are, respectively, the task graph and the processor graph introduced in Sections A.1 and A.3.

- One user task user.task for each task $t \in N_T$. The number of such tasks allocated to run on processor $p \in N_P$ is n.tasks[p] (a variable i indicates one of them). The variable p contains the identity of p.

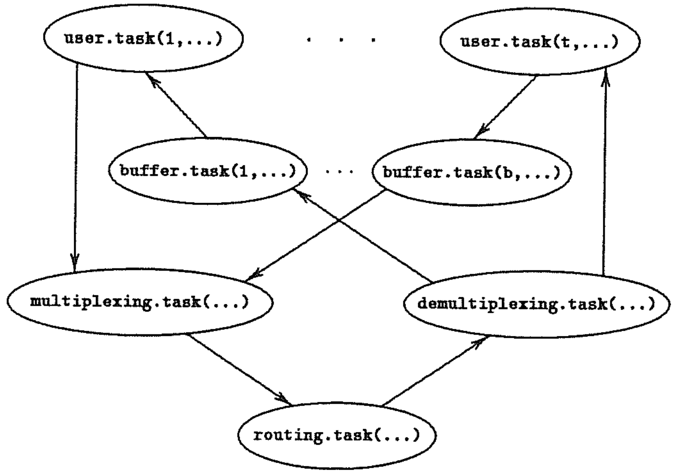


Figure B.2. *This arrangement of Occam tasks summarizes a generic approach to Occam program design. It includes, in addition to the tasks that constitute the user's program proper, tasks for message buffering, message multiplexing, message demultiplexing, and message routing. As indicated in the figure, message buffering can be implemented either at the origin-end or at the destination-end of the task-to-task communication path.*

- One buffer task buffer.task for each set of buffers. The number of such tasks allocated to run on processor $p \in N_P$ is n.buffer.sets[p] (a variable i indicates one of them). This number depends on how the buffers are implemented (cf. Section B.1) and on which end processor the set of buffers corresponding to a same channel in E_T are allocated to when the channel interconnects tasks allocated to distinct processors. The variable p contains the identity of p.
- One routing task routing.task for each processor $p \in N_P$. For generality, this task has only one input channel and one output channel for communication with the locally residing tasks.
- One multiplexing task multiplexing.task for each processor $p \in N_P$. This task receives from every task residing locally whatever message they wish to send to another processor, and then sends it to routing.task.
- One demultiplexing task demultiplexing.task for each processor $p \in N_P$. This task distributes among the tasks that reside locally the messages

that it receives from `routing.task`.

The Occam program to run on processor $p \in N_P$ has the following general appearance, where the tasks we just listed are treated as procedures.

```
PRI PAR
  PAR
    PAR i = 1 FOR n.buffer.sets[p]
      buffer.task(i,...)
    multiplexing.task(...)
    demultiplexing.task(...)
    routing.task(...)
  SEQ
    PAR i = 1 FOR n.tasks[p]
      user.task(i,...)
    PAR
      PAR i = 1 FOR n.buffer.sets[p]
        to.buffer.task[i]!stop
      to.multiplexing.task!stop
      to.demultiplexing.task!stop
      to.routing.task!stop
```

In giving this fragment of an Occam program, we have omitted a considerable amount of information, but it should be meaningful in the light of our discussion in Section B.1. It requires explanations, however, especially concerning the termination of the program at p and throughout the system. This termination is achieved with the aid of additional channels directed to each `buffer.task`, to `multiplexing.task`, to `demultiplexing.task`, and to `routing.task`. These additional channels are called, respectively, `to.buffer.task[i]` (where `i` indicates one of the `n.buffer.sets[p]` tasks `buffer.task`), `to.multiplexing.task`, `to.demultiplexing.task`, and `to.routing.task`.

The program consists of a `PRI PAR` with two components, the one with highest priority to execute the auxiliary tasks (buffer, routing, multiplexing and demultiplexing tasks) in parallel, and the other to first execute in parallel the user tasks and then send a `stop` message on the additional channels to the auxiliary tasks. Note that, in order for `stop` messages to be sent, two conditions must be met. First, every user task residing locally must have terminated (this is decided at the level of graph G_T by one of the techniques seen in Section 3.3 for global termination detection), by the semantics of `SEQ`. Secondly, every auxiliary task must be suspended, by the semantics of `PRI PAR`. Of these suspended tasks, the buffer tasks, the multiplexing task, and the demultiplexing task can be terminated (as the user tasks, already terminated, no longer require their services), and do so upon receiving the `stop`. The routing task, on the other hand, may still be needed as a relay on some path between two other processors, and cannot therefore simply terminate upon receiving the `stop`. What the `stop` does to the routing task is then to trigger a global termination detection algorithm (like the ones seen in Section 3.3), this time at the level of G_P, in which all routing tasks participate for eventual global termination.

B.3. AUTOMATON NETWORK SIMULATION IN OCCAM

Occam programs for the distributed parallel algorithms for automaton network simulation discussed throughout the book can be developed following the general template given in Section B.2, with the adoption of some simplifications aimed at reducing the number of processes per processor, as in current versions of the transputer the cost of process scheduling is still significant. The following are the two main changes to the general template.

- Only one user task per processor is employed. This user task is responsible for various nodes of $G = (N, E)$, the automaton network graph introduced in Chapter 1, as dictated by the procedure for task allocation (data partitioning, in fact). With this simplification, the need for the multiplexing and demultiplexing tasks is eliminated. In addition, buffers are now needed only for communication across processor boundaries (the others can be turned into variables local to the single user task).
- All buffers are allocated to the destination processor and grouped under the responsibility of a single set of two buffer tasks per processor, as explained in Section B.1. This set of buffer tasks receives messages from the routing task and sends them on to the user task. Outputs from the user task are sent directly to the routing task.

The overall appearance of the resulting Occam program is then as we give next. This Occam code is derived directly from the one given in Section B.2 as a general template by performing the simplifications we have just discussed.

```
PRI PAR
  PAR
    buffer.task(...)
    routing.task(...)
  SEQ
    user.task(...)
    PAR
      to.buffer.task!stop
      to.routing.task!stop
```

B.4. BIBLIOGRAPHIC NOTES

The Occam programming language, surveyed briefly in Section B.1, is described in detail by Perrott (1987) and Burns (1988). The underlying mechanisms inherited from CSP can be found in Hoare (1978, 1984).

The material in Sections B.2 and B.3 is based on Barbosa, Drummond, and Hellmuth (1991a, 1991b). The approach is based on the Occam simulation of a communication processor (Barbosa and França, 1988; Drummond, 1990), which employs distributed algorithm building blocks as instructions (some of them are from Segall (1983)). The need and importance of such building blocks has been stressed by Gafni (1986). Additional aspects of the overall approach are described in more detail by Hellmuth (1991).

C

Long proofs

Two long proofs of theorems stated earlier in the book are given in this appendix. The proof of Theorem 4.4 is given in Section C.1, while the proof of Theorem 9.1 is given in Section C.2. Bibliographic notes appear in Section C.3.

C.1. PROOF OF THEOREM 4.4

Our proof of Theorem 4.4 depends on a few preliminary definitions and results. We consider here the sequence of acyclic orientations $\omega_1, \omega_2, \ldots$, introduced in Section 4.2.3, in which ω_{k+1} is obtained from ω_k, for $k \geq 1$, by turning every sink in ω_k into a source.

Lemma C.1. *For $k \geq 1$, a sink in ω_{k+1} has at least one neighbor that is a sink in ω_k.*

Proof: No sink in ω_{k+1} is a sink in ω_k, so every sink in ω_{k+1} has at least one neighbor that is a source in ω_{k+1} and is not a source in ω_k. Such a neighbor has to be a sink in ω_k. Thence the lemma. ■

In the following theorem, we state an important relationship between the periodic repetition of orientations and node multicolorings. If for $k \geq 1$ a k-tuple coloring uses the q colors $c_0, \ldots, c_{q-1}$, we denote by C_ℓ the subset of nodes that get color c_ℓ, for $0 \leq \ell \leq q-1$.

Theorem C.2. *Let $\alpha_0, \ldots, \alpha_{p-1}$ be a period in which each node is a sink in m orientations. Then G admits a p-color, m-tuple coloring by colors $c_0, \ldots, c_{p-1}$ such that $C_k = Sinks(\alpha_k)$ for $0 \leq k \leq p-1$. Moreover, the following two properties hold for this coloring.*

(i) Let $c_{i_1}, \ldots, c_{i_m}$ and $c_{j_1}, \ldots, c_{j_m}$ be the m colors assigned to nodes n_i and n_j, respectively, and assume without loss of generality that $i_1 < \cdots < i_m$ and $j_1 < \cdots < j_m$. If n_i and n_j are neighbors such that $\alpha_0(n_i, n_j) = n_i$, then $i_1 < j_1 < i_2 < j_2 < \cdots < i_m < j_m$.

(ii) For $0 \leq k \leq p-1$, each node in C_k has at least one neighbor in $C_{(k-1)\bmod p}$.

Conversely, if G oriented by α_0 admits a p-color, m-tuple coloring by colors $c_0, \ldots, c_{p-1}$ obeying properties (i) and (ii), then α_0 repeats itself according to the sequence $\alpha_0, \ldots, \alpha_{p-1}, \alpha_0$, in which each node is a sink in m of the orientations $\alpha_0, \ldots, \alpha_{p-1}$, and such that, for $0 \leq k \leq p-1$, $Sinks(\alpha_k) = C_k$.

Proof: In order to show the first part, color the nodes in $Sinks(\alpha_k)$ with color c_k, for $0 \leq k \leq p-1$. Each node is a sink in exactly m of the orientations in the period, so receives exactly m colors. This coloring is then a p-color, m-tuple coloring such that $C_k = Sinks(\alpha_k)$ for all applicable values of k. If $\alpha_0(n_i, n_j) = n_i$ for an edge (n_i, n_j), it must be that $i_1 < j_1 < i_2 < j_2 < \cdots < i_m < j_m$, as the orientations in which node n_i is a sink must alternate with those in which node n_j is a sink, and n_i must be the first one to become a sink starting at α_0. Also, it must be, by Lemma C.1, that each node in C_k has at least one neighbor in $C_{(k-1)\bmod p}$ for $0 \leq k \leq p-1$.

Now we show the converse statement. By property (i), all nodes in C_0 are sinks in α_0, i.e., $C_0 \subseteq Sinks(\alpha_0)$. Also, if a node not in C_0 is to be a sink in α_0, then it must be, by property (i), that either it is a member of C_1 and has no neighbor in C_0, or it is a member of C_2 and has no neighbor in $C_0 \cup C_1$, etc., all the way through its being a member of C_{p-1} with no neighbor in $C_0 \cup \cdots \cup C_{p-2}$. In any of these cases, property (ii) is contradicted, and consequently $Sinks(\alpha_0) \subseteq C_0$, that is, $Sinks(\alpha_0) = C_0$. The same argument can be applied inductively to show that $Sinks(\alpha_1) = C_1, \ldots, Sinks(\alpha_p) = C_0$, inasmuch as property (i) guarantees that all nodes in C_k are sinks in α_k, and property (ii) implies that such is the case for nodes in C_k only, where α_k denotes the orientation obtained after all nodes in $C_0, \ldots, C_{k-1}$ have become sinks, in this order, for $1 \leq k \leq p$. Because each node belongs to exactly m of the color classes $C_0, \ldots, C_{p-1}$, it must be that each node becomes a sink m times in going from α_0 to α_p. Consequently, it must be that $\alpha_p = \alpha_0$. ■

We give in Figure C.1 an illustration of Theorem C.2.

Corollary C.3. *Consider orientation α_0, and let its sink decomposition be $S_0, \ldots, S_{\lambda-1}$. This orientation is in a period $\alpha_0, \ldots, \alpha_{p-1}$ in which each node is a sink exactly once if and only if each node in S_0 has at least one neighbor in $S_{\lambda-1}$. In this case, $p = \lambda$ and $Sinks(\alpha_k) = S_k$ for $0 \leq k \leq \lambda - 1$.*

Proof: Immediate from Theorem C.2. ■

Before we proceed, let us recall that a simple cycle in an undirected graph is a cycle in which no node appears more than once.

Corollary C.4. *Let α_0 be an orientation in a period in which each node is a sink exactly once, and denote by $S_0, \ldots, S_{\lambda-1}$ its sink decomposition. If G is not a tree, then it contains, for some $q \geq 1$, a simple cycle with $q\lambda$ nodes, which, shown in order of traversal, are*

$$n_{i_0^1}, \ldots, n_{i_{\lambda-1}^1}, \ldots, n_{i_0^q}, \ldots, n_{i_{\lambda-1}^q},$$

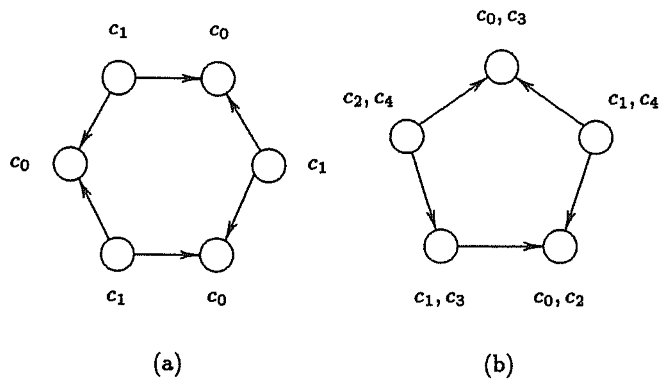


Figure C.1. *The orientations in parts (a) and (b) are both periodic. Shown in this figure are the node multicolorings associated with each of the two orientations. In part (a), a 2-color, 1-tuple coloring is needed, as the period has length two and in it each node becomes a sink once. In part (b), in which the period has length five and in it each node becomes a sink twice, a 5-color, 2-tuple coloring appears.*

where $n_{i_k^\ell} \in S_k$ for all $0 \leq k \leq \lambda - 1$ and all $1 \leq \ell \leq q$.

Proof: Build the simple cycle as follows. Choose a node from those in $S_{\lambda-1}$. By the properties of a sink decomposition, this node must have a neighbor in $S_{\lambda-2}$, this neighbor a neighbor in $S_{\lambda-3}$, and so on. A chain with λ nodes is then obtained, each one belonging to a different component of the sink decomposition. The node in S_0 has a neighbor in $S_{\lambda-1}$, by Corollary C.3, and we pick this node to be part of the chain. If this node is not already in the chain, we repeat the process until a node is picked which is already in the chain. This must happen because of the finiteness of N, except possibly if $\lambda = 2$. In this case, it may happen that a node $n_i \in S_1$ has as only neighbor in S_0 a node n_j, which in turn has n_i as its only neighbor in S_1 (the repetition occurs, but no cycle is formed). Then the search has to be restarted from a node in S_1 which has not yet been visited. A cycle will eventually be found, as G is not a tree. In either case, a simple cycle is formed with $q\lambda$ nodes, for some $q \geq 1$, q of them from each of the components of the sink decomposition. ■

Let K_k be the completely connected graph on k nodes for some $k \geq 1$. In the remainder of this section, we regard the node set of $G[K_k]$, the lexicographic product of G and K_k, as composed of the k layers $L_0, \ldots, L_{k-1}$, each of which is an instance of N (Figure C.2).

Now let α_0 be a periodic orientation in the sequence $\omega_1, \omega_2, \ldots$. We investigate some of the periodicity characteristics of the graph $G\left[K_{m(\omega_1)}\right]$. For such, build an orientation α_0' of $G\left[K_{m(\omega_1)}\right]$ as follows. Edges in the graph formed by one of the

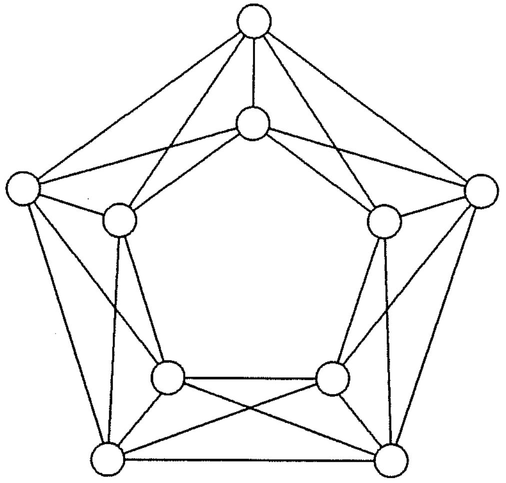


Figure C.2. *This graph is $G[K_2]$, the lexicographic product of G and K_2, when G is the 5-node cycle. The outermost pentagon may be regarded as having L_1 for node set, the innermost one corresponding to L_0.*

layers $L_0, \ldots, L_{m(\omega_1)-1}$ are oriented by α'_0 as those of G are oriented by α_0. Edges between nodes in different layers are oriented by α'_0 from the layer with the highest subscript to the one with the lowest. Consequently, all sources in α'_0 are nodes in $L_{m(\omega_1)-1}$, while all sinks are in L_0. If we let $S'_0, \ldots, S'_{\lambda'-1}$ denote $G[K_{m(\omega_1)}]$'s sink decomposition according to α'_0, then we have $S'_{\lambda'-1} \subseteq L_{m(\omega_1)-1}$ and $S'_0 \subseteq L_0$.

Lemma C.5. *Orientation α'_0 is in a period of length $p(\omega_1)$ in which each node becomes a sink exactly once.*

Proof: By Corollary C.3, it suffices to show that each node in S'_0 has at least one neighbor in $S'_{\lambda'-1}$. Recall from Theorem C.2 that G admits a $p(\omega_1)$-color, $m(\omega_1)$-tuple coloring of its nodes, such that each node with color c_ℓ has at least one neighbor with color $c_{(\ell-1)\bmod p(\omega_1)}$, for $0 \le \ell \le p(\omega_1)-1$. This coloring is such that a node has color c_0 among its colors if and only if it is a sink. A corresponding $p(\omega_1)$-color, 1-tuple coloring for the nodes of $G[K_{m(\omega_1)}]$ can be obtained as follows. A node in $L_{m(\omega_1)-1}$ gets the color with the highest subscript among the $m(\omega_1)$ colors assigned to the corresponding node in G, a node in $L_{m(\omega_1)-2}$ gets the color with the second highest subscript, and so on. Clearly, each node in S'_0 gets color c_0, and must have a neighbor with color $c_{p(\omega_1)-1}$, which must be a node in $L_{m(\omega_1)-1}$. That this node is a member of $S'_{\lambda'-1}$ comes from the fact that in G every node with color $c_{p(\omega_1)-1}$ is a source, by Theorem C.2. ■

It is a consequence of Lemma C.5 that $\lambda' = p(\omega_1)$, as by Corollary C.3 the sets of sinks in the orientations of the period in which α'_0 participates are precisely the components of the sink decomposition of α'_0.

Lemma C.6. *The ratio $\rho(\kappa, \omega_k)$ is the same for all $\kappa \in \mathrm{K}$ and all $k \ge 1$.*

Proof: Immediate from the observation that the reversal of sinks does not change the numbers $n^+(\kappa, \omega_k)$ and $n^-(\kappa, \omega_k)$. This is so because each sink in the cycle has as many edges incident to it which are oriented clockwise by ω_k as it has edges incident to it which ω_k orients counter-clockwise. ■

Proof of Theorem 4.4: Let α_0 be the first periodic orientation in $\omega_1, \omega_2, \ldots$. By Lemma C.6, it suffices to show the assertion of the theorem for α_0. In addition, it suffices, by Theorem 4.3, to show that

$$\frac{m(\omega_1)}{p(\omega_1)} = \min_{\kappa \in \mathrm{K}} \rho(\kappa, \alpha_0).$$

We do this by first showing that $\rho(\kappa, \alpha_0) \ge m(\omega_1)/p(\omega_1)$ for all $\kappa \in \mathrm{K}$, and then showing that

$$\min_{\kappa \in \mathrm{K}} \rho(\kappa, \alpha_0) \le \frac{m(\omega_1)}{p(\omega_1)}.$$

In order to show that $\rho(\kappa, \alpha_0) \ge m(\omega_1)/p(\omega_1)$ for all $\kappa \in \mathrm{K}$, consider the graph G^κ formed by the $|\kappa|$ nodes in κ. This graph is a simple cycle with $|\kappa|$ nodes (several instances of a same node in N may appear in this graph). Let $\#_\kappa(\omega)$ denote the number of sinks in G^κ under orientation ω. Similarly, let $\#_{\kappa,G}(\omega)$ denote the number of sinks in G^κ which are also sinks in G, i.e., members of *Sinks*(ω). Clearly, we have the inequalities

$$\begin{aligned}\rho(\kappa, \omega) &\ge \frac{\#_\kappa(\omega)}{|\kappa|} \\ &\ge \frac{\#_{\kappa,G}(\omega)}{|\kappa|}\end{aligned}$$

for every acyclic orientation ω of G. In particular, if we take the sum on the period $\alpha_0, \ldots, \alpha_{p(\omega_1)-1}$, we get

$$\begin{aligned}\sum_{\alpha \in \{\alpha_0, \ldots, \alpha_{p(\omega_1)-1}\}} \rho(\kappa, \alpha) &\ge \frac{\sum_{\alpha \in \{\alpha_0, \ldots, \alpha_{p(\omega_1)-1}\}} \#_{\kappa,G}(\alpha)}{|\kappa|} \\ &= \frac{m(\omega_1)|\kappa|}{|\kappa|} \\ &= m(\omega_1).\end{aligned}$$

The left-hand side of this inequality is equal to $p(\omega_1)\rho(\kappa, \alpha_k)$ for all $0 \le k \le p(\omega_1)-1$, so we get the desired inequality

$$\rho(\kappa, \alpha_0) \ge \frac{m(\omega_1)}{p(\omega_1)},$$

for all $\kappa \in \mathrm{K}$.

To complete the proof, we must show that

$$\min_{\kappa\in\mathrm{K}} \rho(\kappa,\alpha_0) \leq \frac{m(\omega_1)}{p(\omega_1)}.$$

To do this, we resort to the graph $G[K_{m(\omega_1)}]$ introduced earlier, and to its orientation α_0', which by Lemma C.5 is in a period of length $p(\omega_1)$ in which each node becomes a sink once. By Corollary C.4, a simple cycle exists in $G[K_{m(\omega_1)}]$ with a number of nodes which is multiple of λ' (recall that $S_0,\ldots,S_{\lambda'-1}$ is $G[K_{m(\omega_1)}]$'s sink decomposition according to α_0'). This simple cycle may be traversed as follows. Pick the first node in $S_{\lambda'-1}$, the next one in $S_{\lambda'-2}$, etc., until a node in S_0 is picked, then another one in $S_{\lambda'-1}$, and so on, until the simple cycle is closed. If we extend the definition of the ratio ρ to the graph $G[K_{m(\omega_1)}]$, we immediately see that, if κ' is the cycle we just described, then $\rho(\kappa',\alpha_0') = 1/p(\omega_1)$. This is so because, if the traversal we described is in the clockwise direction, then there exist $p(\omega_1)-1$ consecutive edges in the clockwise direction for each edge encountered in the counter-clockwise direction, as $\lambda' = p$. This simple cycle κ' in $G[K_{m(\omega_1)}]$ corresponds to a cycle κ, not necessarily simple, in G. To see how the value of $\rho(\kappa,\alpha_0)$ behaves, we notice that, as the cycle κ' is traversed as described, a traversal also takes place on the cycle κ. This traversal of κ may only encounter edges oriented in the opposite (i.e., counter-clockwise) direction when the traversal of κ' moves from one of the layers $L_0,\ldots,L_{m(\omega_1)-1}$ to another, because while the traversal is in the same layer the orientations of edges in κ and κ' are the same. Consequently, we see that $\rho(\kappa,\alpha_0) \leq m(\omega_1)/p(\omega_1)$, thus completing the proof. In Figure C.3, we offer an illustration of the second part of this proof. ■

C.2. PROOF OF THEOREM 9.1

Our proof of Theorem 9.1 is based on the following lemma, which gives the so-called *Möbius inversion formulas* in our present context.

Lemma C.7. *Let A be a finite set, and for each $B \subseteq A$ let f_B and g_B be real functions on D^n. Then*

$$f_A(d_1,\ldots,d_n) = \sum_{B\subseteq A} g_B(d_1,\ldots,d_n)$$

if and only if

$$g_A(d_1,\ldots,d_n) = \sum_{B\subseteq A} (-1)^{|A-B|} f_B(d_1,\ldots,d_n)$$

for all $(d_1,\ldots,d_n) \in D^n$.

Proof: If

$$g_A(d_1,\ldots,d_n) = \sum_{B\subseteq A} (-1)^{|A-B|} f_B(d_1,\ldots,d_n),$$

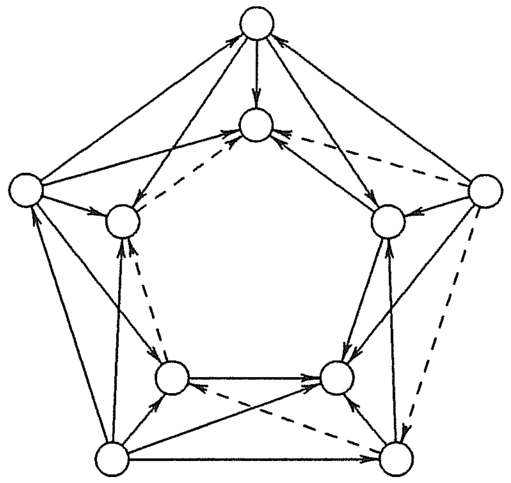


Figure C.3. *This graph is $G[K_2]$, oriented by α_0', when G is the 5-node cycle. Dashed edges correspond to the simple cycle κ'. The outermost pentagon has L_1 for node set, the innermost one corresponding to L_0.*

then

$$\sum_{B\subseteq A} g_B(d_1,\ldots,d_n) = \sum_{B\subseteq A}\sum_{F\subseteq B} (-1)^{|B-F|} f_F(d_1,\ldots,d_n)$$
$$= \sum_{F\subseteq A} f_F(d_1,\ldots,d_n) \sum_{F\subseteq B\subseteq A} (-1)^{|B-F|},$$

where the rightmost summation equals zero unless $F = A$, in which case it equals one and

$$\sum_{B\subseteq A} g_B(d_1,\ldots,d_n) = f_A(d_1,\ldots,d_n).$$

Conversely, if

$$f_A(d_1,\ldots,d_n) = \sum_{B\subseteq A} g_B(d_1,\ldots,d_n),$$

then

$$\begin{aligned}\sum_{B\subseteq A}(-1)^{|A-B|}f_B(d_1,\ldots,d_n) &= \sum_{B\subseteq A}(-1)^{|A-B|}\sum_{F\subseteq B}g_F(d_1,\ldots,d_n)\\ &= \sum_{B\subseteq A}\sum_{F\subseteq B}(-1)^{|A-B|}g_F(d_1,\ldots,d_n)\\ &= \sum_{F\subseteq A}g_F(d_1,\ldots,d_n)\sum_{F\subseteq B\subseteq A}(-1)^{|A-B|},\end{aligned}$$

where the rightmost summation equals zero unless $F = A$, in which case it equals one and

$$\sum_{B\subseteq A}(-1)^{|A-B|}f_B(d_1,\ldots,d_n) = g_A(d_1,\ldots,d_n).$$

■

Proof of Theorem 9.1: We first show that every GRF with respect to the neighborhood $\mathcal{Q}$ and the distribution P is also an MRF with respect to the same neighborhood and the same distribution. For such, consider the Boltzmann-Gibbs distribution of (9.4),

$$P(d_1,\ldots,d_n) = \frac{\exp\bigl(-E(d_1,\ldots,d_n)/T\bigr)}{\sum_{(d'_1,\ldots,d'_n)\in D^n}\exp\bigl(-E(d'_1,\ldots,d'_n)/T\bigr)},$$

where the energy function is given by

$$E(d_1,\ldots,d_n) = \sum_{C\in\mathbf{C}}V_C(d_1,\ldots,d_n).$$

This distribution is clearly strictly positive for all $(d_1,\ldots,d_n)\in D^n$, thereby satisfying (9.1), so it suffices to show that it also satisfies (9.2), i.e., that

$$P(d_i \mid d_j;\ v_j\neq v_i) = P\bigl(d_i \mid d_j;\ v_j\in\mathcal{Q}(v_i)\bigr)$$

for all $v_i\in V$. For such, we first write the energy that may depend on v_i,

$$E_i(d_1,\ldots,d_n) = \sum_{C\in\mathbf{C}\mid v_i\in C}V_C(d_1,\ldots,d_n),$$

and then clearly

$$P(d_i \mid d_j;\ v_j\neq v_i) = \frac{\exp\bigl(-E_i(d_1,\ldots,d_n)/T\bigr)}{\sum_{d'_i\in D}\exp\bigl(-E_i(d_1,\ldots,d'_i,\ldots,d_n)/T\bigr)},$$

where $(d_1,\ldots,d'_i,\ldots,d_n)$ agrees with $(d_1,\ldots,d_n)$ at all variables, except at v_i, which has value d'_i. This first part of the theorem then follows by noticing that

$$P\bigl(d_i,d_j;\ v_j\in\mathcal{Q}(v_i)\bigr) = \alpha\exp\bigl(-E_i(d_1,\ldots,d_n)/T\bigr)$$

and that

$$P\bigl(d_j;\ v_j\in\mathcal{Q}(v_i)\bigr) = \alpha\sum_{d'_i\in D}\exp\bigl(-E_i(d_1,\ldots,d'_i,\ldots,d_n)/T\bigr),$$

where α is the sum, over all combinations of values for variables in $V-\{v_i\}-\mathcal{Q}(v_i)$, of $\exp(-E+E_i)$ at those combinations, so

$$\begin{aligned}P\bigl(d_i \mid d_j;\ v_j\in\mathcal{Q}(v_i)\bigr) &= \frac{P\bigl(d_i,d_j;\ v_j\in\mathcal{Q}(v_i)\bigr)}{P\bigl(d_j;\ v_j\in\mathcal{Q}(v_i)\bigr)}\\ &= \frac{\exp\bigl(-E_i(d_1,\ldots,d_n)/T\bigr)}{\sum_{d'_i\in D}\exp\bigl(-E_i(d_1,\ldots,d'_i,\ldots,d_n)/T\bigr)}.\end{aligned}$$

We now show that every MRF with respect to $\mathcal{Q}$ and P is also a GRF with respect to $\mathcal{Q}$ and P. This is done by displaying an energy function E such that P is the Boltzmann-Gibbs distribution of (9.4). In constructing such a function, we must ensure that each of the potentials V_C involved is nonzero only if in C every two variables are neighbors, as required by the definition of the Boltzmann-Gibbs distribution. For $(d_1,\ldots,d_n)\in D^n$ and $A\subseteq V$, and assuming without any loss in generality that $0\in D$, let

$$d_i^A = \begin{cases} d_i, & \text{if } v_i\in A;\\ 0, & \text{if } v_i\notin A\end{cases}$$

for all $v_i\in V$, and consider the potential

$$V_A(d_1,\ldots,d_n) = \begin{cases} -\ln\bigl(ZP(d_1=0,\ldots,d_n=0)\bigr), & \text{if } A=\emptyset;\\ -\sum_{B\subseteq A}(-1)^{|A-B|}\ln P(d_i^B \mid d_j^B;\ v_j\neq v_i), & \text{if } A\neq\emptyset\end{cases}$$

for some $Z>0$, where v_i is any variable in A. (At this moment, what matters about $-\ln\bigl(ZP(d_1=0,\ldots,d_n=0)\bigr)$ is that it is a constant.) This is a legitimate potential, as by (9.1) P is strictly positive all over D^n. If $|A|>1$ and in A at least two variables are not neighbors, then let v_i and v_k be variables in A such that $v_k\notin\mathcal{Q}(v_i)$. We then have, adopting the abbreviation $B+i$ for $B\cup\{v_i\}$,

$$\begin{aligned}V_A(d_1,\ldots,d_n) &= -\sum_{B\subseteq A}(-1)^{|A-B|}\ln P(d_i^B \mid d_j^B;\ v_j\neq v_i)\\ &= -\sum_{B\subseteq A-\{v_i,v_k\}}\Bigl((-1)^{|A-B|}\ln P(d_i^B \mid d_j^B;\ v_j\neq v_i)\\ &\qquad +(-1)^{|A-B|-1}\ln P(d_i^{B+i} \mid d_j^{B+i};\ v_j\neq v_i)\\ &\qquad +(-1)^{|A-B|-1}\ln P(d_i^{B+k} \mid d_j^{B+k};\ v_j\neq v_i)\\ &\qquad +(-1)^{|A-B|-2}\ln P(d_i^{B+i+k} \mid d_j^{B+i+k};\ v_j\neq v_i)\Bigr)\\ &= -\sum_{B\subseteq A-\{v_i,v_k\}}(-1)^{|A-B|}\\ &\qquad \ln\left(\frac{P(d_i^B \mid d_j^B;\ v_j\neq v_i)/P(d_i^{B+k} \mid d_j^{B+k};\ v_j\neq v_i)}{P(d_i^{B+i} \mid d_j^{B+i};\ v_j\neq v_i)/P(d_i^{B+i+k} \mid d_j^{B+i+k};\ v_j\neq v_i)}\right).\end{aligned}$$

Because $v_k \notin \mathcal{Q}(v_i)$, and by (9.2), we have

$$\begin{aligned}P(d_i^B \mid d_j^B;\ v_j \neq v_i) &= P\big(d_i^B \mid d_j^B;\ v_j \in \mathcal{Q}(v_i)\big)\\ &= P\big(d_i^{B+k} \mid d_j^{B+k};\ v_j \in \mathcal{Q}(v_i)\big)\\ &= P(d_i^{B+k} \mid d_j^{B+k};\ v_j \neq v_i)\end{aligned}$$

and

$$\begin{aligned}P(d_i^{B+i} \mid d_j^{B+i};\ v_j \neq v_i) &= P\big(d_i^{B+i} \mid d_j^{B+i};\ v_j \in \mathcal{Q}(v_i)\big)\\ &= P\big(d_i^{B+i+k} \mid d_j^{B+i+k};\ v_j \in \mathcal{Q}(v_i)\big)\\ &= P(d_i^{B+i+k} \mid d_j^{B+i+k};\ v_j \neq v_i),\end{aligned}$$

so

$$V_A(d_1,\ldots,d_n) = 0.$$

Having shown that the proposed potentials are zero over D^n for subsets of V in which at least two variables are not neighbors, it remains only to argue that they yield a Boltzmann-Gibbs distribution for P. The first step is to rewrite the potentials V_A for nonempty A as

$$\begin{aligned}V_A(d_1,\ldots,d_n) &= -\sum_{B\subseteq A}(-1)^{|A-B|}\ln P(d_i^B \mid d_j^B;\ v_j\neq v_i)\\ &= -\sum_{B\subseteq A-\{v_i\}}\Big((-1)^{|A-B|}\ln P(d_i^B \mid d_j^B;\ v_j\neq v_i)\\ &\qquad + (-1)^{|A-B|-1}\ln P(d_i^{B+i} \mid d_j^{B+i};\ v_j\neq v_i)\Big)\\ &= -\sum_{B\subseteq A-\{v_i\}}(-1)^{|A-B|}\ln\left(\frac{P(d_i^B \mid d_j^B;\ v_j\neq v_i)}{P(d_i^{B+i} \mid d_j^{B+i};\ v_j\neq v_i)}\right)\\ &= -\sum_{B\subseteq A-\{v_i\}}(-1)^{|A-B|}\\ &\qquad \ln\left(\frac{P(d_1^B,\ldots,d_n^B)/P(d_j^B;\ v_j\neq v_i)}{P(d_1^{B+i},\ldots,d_n^{B+i})/P(d_j^B;\ v_j\neq v_i)}\right)\\ &= -\sum_{B\subseteq A-\{v_i\}}(-1)^{|A-B|}\ln\left(\frac{P(d_1^B,\ldots,d_n^B)}{P(d_1^{B+i},\ldots,d_n^{B+i})}\right)\\ &= -\sum_{B\subseteq A-\{v_i\}}\Big((-1)^{|A-B|}\ln P(d_1^B,\ldots,d_n^B)\\ &\qquad + (-1)^{|A-B|-1}\ln P(d_1^{B+i},\ldots,d_n^{B+i})\Big)\\ &= -\sum_{B\subseteq A}(-1)^{|A-B|}\ln P(d_1^B,\ldots,d_n^B),\end{aligned}$$

and then recognize that they may be defined equivalently as

$$V_A(d_1,\ldots,d_n) = -\sum_{B\subseteq A}(-1)^{|A-B|}\ln\big(ZP(d_1^B,\ldots,d_n^B)\big)$$

for all $(d_1,\ldots,d_n)\in D^n$ and some $Z>0$, inasmuch as

$$\sum_{B\subseteq A}(-1)^{|A-B|} = 0.$$

By Lemma C.7 with

$$f_A(d_1,\ldots,d_n) = -\ln\big(ZP(d_1^A,\ldots,d_n^A)\big)$$

and

$$g_A(d_1,\ldots,d_n) = V_A(d_1,\ldots,d_n)$$

for all $(d_1,\ldots,d_n)\in D^n$, we then have

$$-\ln\big(ZP(d_1^A,\ldots,d_n^A)\big) = \sum_{B\subseteq A}V_B(d_1,\ldots,d_n).$$

Notice that this holds for $A=\emptyset$ as well, as $d_i^\emptyset = 0$ for all $v_i\in V$ and $V_\emptyset = -\ln\big(ZP(d_1=0,\ldots,d_n=0)\big)$. For $A=V$, and considering that $V_B=0$ whenever in B at least two variables are not neighbors, we obtain

$$P(d_1,\ldots,d_n) = \frac{1}{Z}\exp\left(-\sum_{C\in\mathbf{C}}V_C(d_1,\ldots,d_n)\right),$$

which, with

$$E(d_1,\ldots,d_n) = \sum_{C\in\mathbf{C}}V_C(d_1,\ldots,d_n)$$

and

$$Z = \sum_{(d_1',\ldots,d_n')\in D^n}\exp\big(-E(d_1',\ldots,d_n')/T\big),$$

yields

$$P(d_1,\ldots,d_n) = \frac{\exp\big(-E(d_1,\ldots,d_n)/T\big)}{\sum_{(d_1',\ldots,d_n')\in D^n}\exp\big(-E(d_1',\ldots,d_n')/T\big)},$$

the desired Boltzmann-Gibbs distribution. ■

C.3. BIBLIOGRAPHIC NOTES

The proof of Theorem 4.4 given in Section C.1 is from Barbosa and Gafni (1987, 1989b). The notions of node multicolorings in graphs can also be looked up in Stahl (1976).

Our proof of Theorem 9.1 in Section C.2 follows Griffeath (1976), and appears to have been first given by Grimmett (1973). The reference for the Möbius inversion formulas and related material is Rota (1964).

D

Additional edge-reversal properties

This appendix contains, in Section D.1, a discussion of some aspects of scheduling by edge reversal left uncovered in Chapter 4. Bibliographic notes are given after that in Section D.2.

D.1. REACHABILITY EQUATIONS

The general scheme we saw in Section 4.3.3 for the simulation of PC automaton networks is in two cases (Markov random fields and Bayesian networks) a variation of the scheduling by edge reversal mechanism of Section 4.2 in which all nodes are required to be updated a same number (K) of times. This variation goes along the following general lines. A processor performs its updatings whenever its node becomes a sink, and does it for the last time when it is the Kth time it has done it. At this time, it simply waits for one reversal message from each of its neighbors that are responsible for its node's upstream neighbors in the initial acyclic orientation (which then become upstream again). We provide in this section a proof that every node becomes a sink exactly K times, regardless of the initial orientation, and that at the end of the computation the initial acyclic orientation is indeed restored.

Our argument is based on the following generalization of the scheduling by edge reversal mechanism, which, as in Section 4.2.3, we describe under the assumption of a synchronous model of distributed computation. At each clock pulse, at least one of the sinks is turned into a source, but not necessarily all of them. We call any sequence of orientations thus generated a ***schedule of orientations***, or simply a ***schedule***. In a schedule $\omega_1, \omega_2, \ldots$, we say that ω_ℓ is ***reachable*** from ω_k, for $k, \ell \geq 1$, if and only if $\ell > k$. Our goal in this section is to show that every orientation is reachable from itself along any portion of a schedule in which all nodes become sinks and turn into sources the same number of times, and only thereby (the period of orientations studied in Section 4.2.3 represents then simply a particular case of this property, as by Corollary 4.2 all nodes become sinks the same number of times in a period). In this way, the correctness of the aforementioned variation of the scheduling by edge reversal mechanism will have been established.

Let ω and ω' be any two acyclic orientations of G, and denote by $m_i(\omega,\omega')$ the number of times that a node n_i has to be turned from a sink into a source in order to generate ω' through some schedule starting at ω. Say that an edge (n_i,n_j) is *marked* with respect to ω and ω', or simply *marked*, if and only if $\omega(n_i,n_j) \neq \omega'(n_i,n_j)$. For all edges (n_i,n_j) such that $\omega(n_i,n_j)=n_j$, the *reachability equations* from ω to ω' are

$$m_j(\omega,\omega') = \begin{cases} m_i(\omega,\omega')+1, & \text{if } (n_i,n_j) \text{ is marked;} \\ m_i(\omega,\omega'), & \text{otherwise.} \end{cases}$$

Theorem D.1. *For any two acyclic orientations ω and ω' of G, the reachability equations from ω to ω' admit a nonnegative integer solution if and only if ω' is reachable from ω.*

Proof: If ω' is reachable from ω through some schedule, then let y_i denote the number of times n_i is turned from sink into source from ω to ω' in that schedule. If an edge (n_i,n_j) such that $\omega(n_i,n_j)=n_j$ is marked, then in the schedule n_j has to be turned from sink into source exactly once more than n_i has from ω to ω', i.e., $y_j = y_i + 1$. If the edge is not marked, then both n_i and n_j have to be turned from sinks into sources in the schedule the same number of times from ω to ω', i.e., $y_j = y_i$. Thus the n-tuple $(y_i;\ n_i \in N)$ is a nonnegative integer solution to the reachability equations from ω to ω'.

Conversely, let $(y_i;\ n_i \in N)$ be a nonnegative integer solution to the reachability equations from ω to ω'. If edge (n_i,n_j) is such that $\omega(n_i,n_j)=n_j$, then it must be that $y_j = y_i+1$ if (n_i,n_j) is marked, and $y_j = y_i$ otherwise. If we build a schedule in which each node n_i is turned from sink into source exactly y_i times up to a certain point, then at that point each marked edge will have been reversed an odd number of times, whereas an edge that is not marked will have been reversed an even number of times. In other words, ω' will have been reached. It remains to argue that such a schedule can always be built. For such, notice that the nonnegative integers $(y_i;\ n_i \in N)$ are nondecreasing along any directed path in G oriented by ω, so at least one sink n_i is such that $y_i \geq 1$. If one such sink in ω is turned into a source yielding orientation ω'', then the same n-tuple, with $y_i - 1$ replacing y_i, is a nonnegative integer solution to the reachability equations from ω'' to ω'. This solution is also nondecreasing along any directed path, and we see as a consequence that a sink can always be found to be turned into a source until ω' is reached. ∎

Corollary D.2. *Any orientation is reachable from itself, and the corresponding solutions to the reachability equations are all (and only) n-tuples with the same nonnegative integer in all entries.*

Proof: Immediate from the definition of reachability equations and from Theorem D.1. ∎

D.2. BIBLIOGRAPHIC NOTES

This appendix is based on Barbosa (1986).

Bibliography

Abu-Mostafa, Y., and D. Schweizer (1990). Neural networks. In R. Suaya and G. Birtwistle (Eds.), *VLSI and Parallel Computation*, 390–415. Morgan Kaufmann, San Mateo, CA, USA.

Ackley, D. H., G. E. Hinton, and T. J. Sejnowski (1985). A learning algorithm for Boltzmann machines. *Cognitive Science* **9**, 147–169.

Ahuja, R. K., T. L. Magnanti, and J. B. Orlin (1989). Network flows. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd (Eds.), *Handbooks in Operations Research and Management Science, Vol. 1: Optimization*, 211–369. North-Holland, Amsterdam, The Netherlands.

Akiyama, Y., A. Yamashita, M. Kajiura, and H. Aiso (1989). Combinatorial optimization with Gaussian machines. In *Proc. of the International Joint Conference on Neural Networks*, I-533–540.

Akl, S. G. (1989). *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Allen, J. (1987). *Natural Language Understanding.* Benjamin/Cummings, Menlo Park, CA, USA.

Almasi, G. S., and A. Gottlieb (1989). *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, USA.

Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Menlo Park, CA, USA.

Andrews, G. R., and F. B. Schneider (1983). Concepts and notations for concurrent programming. *ACM Computing Surveys* **15**, 3–43.

Angluin, D. (1980). Local and global properties in networks of processors. In *Proc. of the Annual ACM Symposium on Theory of Computing*, 82–93.

Aragon, C. R., D. S. Johnson, L. A. McGeoch, and C. Schevon (1984). Optimization by simulated annealing: an experimental evaluation. *Workshop on Statistical Physics in Engineering and Biology.*

Arjomandi, E., M. J. Fischer, and N. A. Lynch (1983). Efficiency of synchronous versus asynchronous distributed systems. *J. of the ACM* **30**, 449–456.

Arlauskas, S. (1988). iPSC/2 system: a second generation hypercube. In *Proc. of the Conference on Hypercube Concurrent Computers and Applications, Vol. 1*, 38–42.

Attiya, H., and M. Snir (1991). Better computing on the anonymous ring. *J. of Algorithms* **12**, 204–238.

Attiya, H., M. Snir, and M. K. Warmuth (1988). Computing on an anonymous ring. *J. of the ACM* **35**, 845–875.

Awerbuch, B. (1985a). Complexity of network synchronization. *J. of the ACM* **32**, 804–823.

Awerbuch, B. (1985b). Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization. *Networks* **15**, 425–437.

Awerbuch, B. (1987). Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems: detailed summary. In *Proc. of the Annual ACM Symposium on Theory of Computing*, 230–240.

Awerbuch, B., and D. Peleg (1990). Network synchronization with polylogarithmic overhead. In *Proc. of the Annual Symposium on Foundations of Computer Science*, 514–522.

Bacchus, F. (1988). Representing and reasoning with probabilistic knowledge. Ph.D. dissertation, Department of Computer Science, University of Alberta, Edmonton, Canada.

Bacchus, F. (1991). *Representing and Reasoning with Probabilistic Knowledge: A Logical Approach to Probabilities.* The MIT Press, Cambridge, MA, USA.

Barbosa, V. C. (1986). Concurrency in systems with neighborhood constraints. Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, CA, USA.

Barbosa, V. C. (1990a). Strategies for the prevention of communication deadlocks in distributed parallel programs. *IEEE Trans. on Software Engineering* **16**, 1311–1316.

Barbosa, V. C. (1990b). Blocking versus nonblocking interprocess communication: a note on the effect on concurrency. *Information Processing Letters* **36**, 171–175.

Barbosa, V. C. (1991). On the time-driven parallel simulation of two classes of complex systems. Technical report ES-248/91, COPPE/UFRJ, Rio de Janeiro, Brazil.

Barbosa, V. C., and M. C. S. Boeres (1990). An Occam-based evaluation of a parallel version of simulated annealing. In *Proc. of Euromicro-90*, 85–92.

Barbosa, V. C., and L. A. V. de Carvalho (1990). Feasible directions linear programming by neural networks. In *Proc. of the International Joint Conference on Neural Networks*, III-941–946.

Barbosa, V. C., and L. A. V. de Carvalho (1991). Learning in analog Hopfield networks. In *Proc. of the International Joint Conference on Neural Networks*, II-183–186.

Barbosa, V. C., L. M. de A. Drummond, and A. L. H. Hellmuth (1991a). From distributed algorithms to Occam programs by successive refinements. Technical report ES-237/91, COPPE/UFRJ, Rio de Janeiro, Brazil.

Barbosa, V. C., L. M. de A. Drummond, and A. L. H. Hellmuth (1991b). An integrated software environment for large-scale Occam programming. In *Proc. of Euromicro-91*, 393–400.

Barbosa, V. C., and F. M. G. França (1988). Specification of a communication virtual processor for parallel processing systems. In *Proc. of Euromicro-88*, 511–518.

Barbosa, V. C., and E. Gafni (1987). Concurrency in heavily loaded neighborhood-constrained systems. In *Proc. of the International Conference on Distributed Computing Systems*, 448–455.

Barbosa, V. C., and E. Gafni (1989a). A distributed implementation of simulated annealing. *J. of Parallel and Distributed Computing* **6**, 411–434.

Barbosa, V. C., and E. Gafni (1989b). Concurrency in heavily loaded neighborhood-constrained systems. *ACM Trans. on Programming Languages and Systems* **11**, 562–584.

Barbosa, V. C., and H. K. Huang (1988). Static task allocation in heterogeneous distributed systems. Technical report ES-149/88, COPPE/UFRJ, Rio de Janeiro, Brazil.

Barbosa, V. C., and P. M. V. Lima (1990). On the distributed parallel simulation of Hopfield's neural networks. *Software — Practice and Experience* **20**, 967–983.

Bell, G. (1992). Ultracomputers: a teraflop before its time. *Comm. of the ACM* **35**, 26–47.

Ben-Ari, M. (1982). *Principles of Concurrent Programming.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Berge, C. (1976). *Graphs and Hypergraphs.* North-Holland, Amsterdam, The Netherlands.

Bertsekas, D. P., and R. G. Gallager (1987). *Data Networks.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Bertsekas, D. P., and J. N. Tsitsiklis (1989). *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems. *J. of the Royal Statistical Society, Series B* **36**, 192–236.

Boeres, M. C. S., L. A. V. de Carvalho, and V. C. Barbosa (1992). A faster elastic-net algorithm for the traveling salesman problem. In *Proc. of the International Joint Conference on Neural Networks*, II-215–220.

Bondy, J. A., and U. S. R. Murty (1976). *Graph Theory with Applications.* North-Holland, New York, NY, USA.

Bracha, G., and S. Toueg (1984). A distributed algorithm for generalized deadlock detection. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 285–301.

Brandt, R. D., Y. Wang, A. J. Laub, and S. K. Mitra (1988). Alternative networks for solving the travelling salesman problem and the list-matching problem. In *Proc. of the IEEE International Conference on Neural Networks*, II-333–340.

Braun, M. (1978). *Differential Equations and Their Applications.* Springer-Verlag, New York, NY, USA.

Burns, A. (1988). *Programming in Occam2.* Addison-Wesley, Wokingham, England.

Burr, D. J. (1988). An improved elastic net method for the traveling salesman problem. In *Proc. of the IEEE International Conference on Neural Networks*, I-69–76.

Chandy, K. M., and L. Lamport (1985). Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computer Systems* **3**, 63–75.

Chandy, K. M., and J. Misra (1984). The drinking philosophers problem. *ACM Trans. on Programming Languages and Systems* **6**, 632–646.

Chandy, K. M., and J. Misra (1988). *Parallel Program Design: a Foundation.* Addison-Wesley, Reading, MA, USA.

Chang, E. (1980). n philosophers: an exercise in distributed control. *Computer Networks* **4**, 71–76.

Charniak, E. (1983). Passing markers: a theory of contextual influence in language comprehension. *Cognitive Science* **7**, 171–190.

Charniak, E., and R. Goldman (1989). A semantics for probabilistic quantifier-free first-order languages, with particular application to story understanding. In *Proc. of the International Joint Conference on Artificial Intelligence*, 1074–1079.

Chin, F., and H. F. Ting (1990). Improving the time complexity of message-optimal distributed algorithms for minimum-weight spanning trees. *SIAM J. on Computing* **19**, 612–626.

Chou, C.-T., and E. Gafni (1988). Understanding and verifying distributed algorithms using stratified decomposition. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 44–65.

Christofides, N. (1976). Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, GSIA, Carnegie-Mellon University, Pittsburgh, PA, USA.

Collins, N. E., R. W. Eglese, and B. L. Golden (1988). Simulated annealing: an annotated bibliography. *American J. of Mathematical and Management Sciences* **8**, 209–307.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proc. of the Annual ACM Symposium on Theory of Computing*, 151–158.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence* **42**, 393–405.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest (1990). *Introduction to Algorithms.* The MIT Press, Cambridge, MA, USA.

Cottrell, G. W. (1985). A connectionist approach to word sense disambiguation. Ph.D. dissertation, Department of Computer Science, University of Rochester, Rochester, NY, USA.

Dally, W. J. (1990). Network and processor architecture for message-driven computers. In R. Suaya and G. Birtwistle (Eds.), *VLSI and Parallel Computation*, 140–222. Morgan Kaufmann, San Mateo, CA, USA.

Dally, W. J., L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills (1987). Architecture of a message-driven processor. In *Proc. of the Annual International Symposium on Computer Architecture*, 189–196.

Dally, W. J., and C. L. Seitz (1987). Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers* **C-36**, 547–553.

Dantzig, G. B. (1963). *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, USA.

de Carvalho, L. A. V., and V. C. Barbosa (1989). Towards a stochastic neural model for combinatorial optimization. Technical report ES-196/89, COPPE/UFRJ, Rio de Janeiro, Brazil.

de Carvalho, L. A. V., and V. C. Barbosa (1990). A TSP objective function that ensures feasibility at stable points. In *Proc. of the International Neural Network Conference*, 249–253.

de Carvalho, L. A. V., and V. C. Barbosa (1992). Fast linear system solution by neural networks. *Operations Research Letters* **11**, 141–145.

Dijkstra, E. W. (1968). Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages*, 43–112. Academic Press, New York, NY, USA.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Comm. of the ACM* **18**, 453–457.

Dijkstra, E. W., and C. S. Scholten (1980). Termination detection for diffusing computations. *Information Processing Letters* **11**, 1–4.

Drummond, L. M. de A. (1990). Design and implementation of a communication virtual processor. M.Sc. thesis, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, Brazil. In Portuguese.

Durbin, R., R. Szeliski, and A. Yuille (1989). An analysis of the elastic net approach to the traveling salesman problem. *Neural Computation* **1**, 348–358.

Durbin, R., and D. Willshaw (1987). An analogue approach to the travelling salesman problem using an elastic net method. *Nature* **326**, 689–691.

Edmonds, J. (1965). Maximum matching and a polyhedron with 0, 1-vertices. *J. of Research of the National Bureau of Standards* **69B**, 125–130.

Eizirik, L. M. R. (1990). A Bayesian-network approach to the solution of lexical ambiguities. D.Sc. dissertation, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, Brazil. In Portuguese.

Eizirik, L. M. R., V. C. Barbosa, and S. B. T. Mendes (1993). A Bayesian-network approach to lexical disambiguation. *Cognitive Science*, to appear.

Even, S. (1979). *Graph Algorithms.* Computer Science Press, Potomac, MD, USA.

Fanty, M. (1985). Context-free parsing in connectionist networks. Technical report 174, Computer Science Department, University of Rochester, Rochester, NY, USA.

Feldman, Y., and E. Shapiro (1992). Spatial machines: a more realistic approach to parallel computation. *Comm. of the ACM* **35**, 60–73.

Feller, W. (1968). *An Introduction to Probability Theory and its Applications, Vol. 1.* Wiley, New York, NY, USA.

Feller, W. (1971). *An Introduction to Probability Theory and its Applications, Vol. 2.* Wiley, New York, NY, USA.

Felten, E., S. Karlin, and S. W. Otto (1985). The traveling salesman problem on a hypercubic, MIMD computer. In *Proc. of the International Conference on Parallel Processing*, 6–10.

Fillmore, C. J. (1968). The case for case. In E. Bach and R. T. Harms (Eds.), *Universals in Linguistic Theory*, 1–88. Holt, Rinehart, and Winston, New York, NY, USA.

Fiorini, S., and R. J. Wilson (1977). *Edge-Colourings of Graphs*. Pitman, London, England.

Ford, Jr., L. R., and D. R. Fulkerson (1962). *Flows in Networks*. Princeton University Press, Princeton, NJ, USA.

Fox, G. C., and W. Furmanski (1988). Load balancing loosely synchronous problems with a neural network. In *Proc. of the Conference on Hypercube Concurrent Computers and Applications, Vol. 1*, 241–278.

Fox, G. C., M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker (1988). *Solving Problems on Concurrent Processors, Vol. 1*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Fox, G. C., A. Kolawa, and R. Williams (1987). The implementation of a dynamic load balancer. In M. T. Heath (Ed.), *Hypercube Multiprocessors 1987*, 114–121. SIAM, Philadelphia, PA, USA.

Fox, G. C., and S. W. Otto (1986). Concurrent computation and the theory of complex systems. In M. T. Heath (Ed.), *Hypercube Multiprocessors 1986*, 244–268. SIAM, Philadelphia, PA, USA.

Freitas, L. de P. Sá, and V. C. Barbosa (1991). Experiments in parallel heuristic search. In *Proc. of the International Conference on Parallel Processing*, III-62–65.

Fulkerson, D. R. (1972). Anti-blocking polyhedra. *J. of Combinatorial Theory, Series B* **12**, 50–71.

Gafni, E. (1986). Perspectives on distributed network protocols: a case for building blocks. In *Proc. of Milcom-86*.

Gafni, E., and V. C. Barbosa (1986). Optimal snapshots and the maximum flow in precedence graphs. In *Proc. of the Allerton Conference on Communication, Control, and Computing*, 1089–1097.

Gafni, E. M., and D. P. Bertsekas (1981). Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Trans. on Communications* **COM-29**, 11–18.

Gallager, R. G., P. A. Humblet, and P. M. Spira (1983). A distributed algorithm for minimum weight spanning trees. *ACM Trans. on Programming Languages and Systems* **5**, 66–77.

Garey, M. R., and D. S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, NY, USA.

Geiger, D., T. Verma, and J. Pearl (1990). *d*-separation: from theorems to algorithms. In M. Henrion, R. D. Shachter, L. N. Kanal, and J. F. Lemmer (Eds.), *Uncertainty in Artificial Intelligence 5*, 139–148. North-Holland, Amsterdam, The Netherlands.

Geman, S., and D. Geman (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **PAMI-6**, 721–741.

Gentleman, W. M. (1981). Message passing between sequential processes: the reply primitive and the administrator concept. *Software — Practice and Experience* **11**, 435–466.

Gerla, M., and L. Kleinrock (1982). Flow control protocols. In P. E. Green, Jr. (Ed.), *Computer Network Architectures and Protocols*, 361–412. Plenum Press, New York, NY, USA.

Gibbons, A., and W. Rytter (1988). *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England.

Goldberg, A. V., E. Tardos, and R. E. Tarjan (1990). Network flow algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver (Eds.), *Algorithms and Combinatorics, Vol. 9*, 101–164. Springer-Verlag, Berlin, Germany.

Goldberg, A. V., and R. E. Tarjan (1986). A new approach to the maximum flow problem. In *Proc. of the Annual ACM Symposium on Theory of Computing*, 136–146.

Goldberg, A. V., and R. E. Tarjan (1988). A new approach to the maximum-flow problem. *J. of the ACM* **35**, 921–940.

Goldman, R. P., and E. Charniak (1992). Probabilistic text understanding. *Statistics and Computing* **2**, 105–114.

Goles, E., and S. Martínez (1990). *Neural and Automata Networks*. Kluwer, Dordrecht, The Netherlands.

Golub, G. H., and C. F. van Loan (1983). *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA.

Gonzaga, C. C. (1992). Path-following methods for linear programming. *SIAM Review* **34**, 167–224.

Greening, D. R. (1990). Parallel simulated annealing techniques. *Physica D* **42**, 293–306.

Greening, D. R. (1991). Asynchronous parallel simulated annealing. In L. Nadel and D. L. Stein (Eds.), *1990 Lectures in Complex Systems*, 497–507. Addison-Wesley, Redwood City, CA, USA.

Griffeath, D. (1976). Introduction to random fields. In J. G. Kennedy, J. L. Snell, and A. W. Knapp (Eds.), *Denumerable Markov Chains*, 425–458. Springer-Verlag, New York, NY, USA.

Grimmett, G. R. (1973). A theorem about random fields. *Bull. of the London Mathematical Society* **5**, 81–84.

Grötschel, M., L. Lovász, and A. Schrijver (1981). The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* **1**, 169–197.

Grunwald, D. C., and D. A. Reed (1988). Networks for parallel processors: measurements and prognostications. In *Proc. of the Conference on Hypercube Concurrent Computers and Applications, Vol. 1*, 600–608.

Günther, K. D. (1981). Prevention of deadlocks in packet-switched data transport systems. *IEEE Trans. on Communications* **COM-29**, 512–524.

Harary, F. (1969). *Graph Theory*. Addison-Wesley, Reading, MA, USA.

Hecht-Nielsen, R. (1990). *Neurocomputing*. Addison-Wesley, Reading, MA, USA.

Hegde, S. U., J. L. Sweet, and W. B. Levy (1988). Determination of parameters in a Hopfield/Tank computational network. In *Proc. of the IEEE International*

Conference on Neural Networks, II-291–298.

Hellmuth, A. L. H. (1991). Tools for the development of distributed parallel programs. M.Sc. thesis, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, Brazil. In Portuguese.

Henrion, M. (1990). An introduction to algorithms for inference in belief nets. In M. Henrion, R. D. Shachter, L. N. Kanal, and J. F. Lemmer (Eds.), *Uncertainty in Artificial Intelligence 5*, 129–138. North-Holland, Amsterdam, The Netherlands.

Herskovits, J. (1986). A two-stage feasible directions algorithm for nonlinear constrained optimization. *Mathematical Programming* **36**, 19–38.

Hertz, J., A. Krogh, and R. G. Palmer (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA, USA.

Hillis, W. D. (1985). *The Connection Machine*. The MIT Press, Cambridge, MA, USA.

Hinton, G. E., T. J. Sejnowski, and D. H. Ackley (1984). Boltzmann machines: constraint satisfaction networks that learn. Technical report CMU-CS-84-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, USA.

Hoare, C. A. R. (1978). Communicating sequential processes. *Comm. of the ACM* **21**, 666–677.

Hoare, C. A. R. (1984). *Communicating Sequential Processes*. Prentice-Hall, London, England.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences USA* **79**, 2554–2558.

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. of the National Academy of Sciences USA* **81**, 3088–3092.

Hopfield, J. J. (1986). Simple neural optimization networks: an A/D converter, a signal decision circuit, and a linear programming circuit. *IEEE Trans. on Circuits and Systems* **CAS-33**, 533–541.

Hopfield, J. J., and D. W. Tank (1985). Neural computations of decisions in optimization problems. *Biological Cybernetics* **52**, 141–152.

Hopfield, J. J., and D. W. Tank (1986). Computing with neural circuits: a model. *Science* **233**, 625–633.

Hrycej, T. (1990). Gibbs sampling in Bayesian networks. *Artificial Intelligence* **46**, 351–363.

Isham, V. (1981). An introduction to spatial point processes and Markov random fields. *International Statistical Review* **49**, 21–43.

JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, USA.

Jen, E. (1986a). Global properties of cellular automata. *J. of Statistical Physics* **43**, 219–242.

Jen, E. (1986b). Invariant strings and pattern-recognizing properties of one-dimensional cellular automata. *J. of Statistical Physics* **43**, 243–265.

Jen, E. (1989). Limit cycles in one-dimensional cellular automata. In D. L. Stein (Ed.), *Lectures in the Sciences of Complexity*, 743–758. Addison-Wesley, Redwood City, CA, USA.

Jiang, T. (1989). The synchronization of nonuniform networks of finite automata. In *Proc. of the Annual Symposium on Foundations of Computer Science*, 376–381.

Johnson, D. S., C. R. Aragon, L. A. McGeoch, and C. Schevon (1989). Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations Research* **37**, 865–892.

Johnson, D. S., C. R. Aragon, L. A. McGeoch, and C. Schevon (1991). Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research* **39**, 378–406.

Karlin, S., and H. M. Taylor (1975). *A First Course in Stochastic Processes*. Academic Press, New York, NY, USA.

Karlin, S., and H. M. Taylor (1981). *A Second Course in Stochastic Processes*. Academic Press, New York, NY, USA.

Karmarkar, N. (1984). A new polynomial time algorithm for linear programming. *Combinatorica* **4**, 373–395.

Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher (Eds.), *Complexity of Computer Computations*, 85–103. Plenum Press, New York, NY, USA.

Karp, R. M., and V. Ramachandran (1990). Parallel algorithms for shared-memory machines. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Vol. A*, 869–941. The MIT Press, Cambridge, MA, USA.

Kermani, P., and L. Kleinrock (1979). Virtual cut-through: a new computer communication switching technique. *Computer Networks* **3**, 267–286.

Kernighan, B. W., and S. Lin (1970). An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical J.* **49**, 291–307.

Khachiyan, L. G. (1979). A polynomial algorithm for linear programming. *Doklady Akad. Nauk USSR* **244**, 1093–1096. Translated in *Soviet Math. Doklady* **20**, 191–194.

Kinderman, R., and J. L. Snell (1980). *Markov Random Fields and their Applications*. American Mathematical Society, Providence, RI, USA.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi (1983). Optimization by simulated annealing. *Science* **220**, 671–680.

Kleinrock, L. (1975). *Queueing Systems, Vol. 1: Theory*. Wiley, New York, NY, USA.

Kohonen, T. (1988). *Self-Organization and Associative Memory*. Springer-Verlag, Berlin, Germany.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Krumme, D. W., K. N. Venkataraman, and G. Cybenko (1986). Hypercube embedding is *NP*-complete. In M. T. Heath (Ed.), *Hypercube Multiprocessors 1986*, 148–157. SIAM, Philadelphia, PA, USA.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM* **21**, 558–565.

Lamport, L., and N. A. Lynch (1990). Distributed computing: models and methods. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Vol. B*, 1156–1199. The MIT Press, Cambridge, MA, USA.

Lawler, E. L. (1976). *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, New York, NY, USA.

Lebowitz, J. L., C. Maes, and E. R. Speer (1990). Probabilistic cellular automata: some statistical mechanical considerations. In E. Jen (Ed.), *1989 Lectures in Complex Systems*, 401–414. Addison-Wesley, Redwood City, CA, USA.

Lee, Y. C. (1989). Neural networks and collective computing. In D. L. Stein (Ed.), *Lectures in the Sciences of Complexity*, 799–813. Addison-Wesley, Redwood City, CA, USA.

Leighton, F. T. (1992). *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, CA, USA.

Lin, S., and B. W. Kernighan (1973). An effective heuristic for the traveling-salesman problem. *Operations Research* **21**, 498–516.

Luenberger, D. G. (1973). *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, Reading, MA, USA.

Luenberger, D. G. (1979). *Introduction to Dynamic Systems.* Wiley, New York, NY, USA.

Lynch, N. A. (1980). Fast allocation of nearby resources in a distributed system. In *Proc. of the Annual ACM Symposium on Theory of Computing*, 70–81.

Lynch, N. A. (1981). Upper bounds for static resource allocation in a distributed system. *J. of Computer and System Sciences* **23**, 254–278.

Lynch, N. A., and M. R. Tuttle (1987). Hierarchical correctness proofs for distributed algorithms. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 137–151.

Lynch, N. A., and K. J. Goldman (1989). Distributed algorithms: lecture notes for 6.852, fall 1988. Research Seminar Series MIT/LCS/RSS 5, Laboratory for Computer Science, MIT, Cambridge, MA, USA.

Ma, P. R., E. Y. S. Lee, and M. Tsuchiya (1982). A task allocation model for distributed computing systems. *IEEE Trans. on Computers* **C-31**, 41–47.

Maekawa, M., A. E. Oldehoeft, and R. R. Oldehoeft (1987). *Operating Systems: Advanced Concepts.* Benjamin/Cummings, Menlo Park, CA, USA.

Manna, Z., and A. Pnueli (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, NY, USA.

McClelland, J. L., and A. H. Kawamoto (1986). Mechanisms of sentence processing: assigning roles to constituents of sentences. In J. L. McClelland, D. E. Rumelhart, and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 2: Psychological and Biological Models*, 272–325. The MIT Press, Cambridge, MA, USA.

Mead, C. (1989). *Analog VLSI and Neural Systems.* Addison-Wesley, Reading, MA, USA.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). Equations of state calculations by fast computing machines. *J. of Chemical Physics* **21**, 1087–1091.

Mitra, D., F. Romeo, and A. Sangiovanni-Vincentelli (1986). Convergence and finite-time behavior of simulated annealing. *Advances in Applied Probability* **18**, 747–771.

Moussouris, J. (1974). Gibbs and Markov random systems with constraints. *J. of Statistical Physics* **10**, 11–33.

Nicol, D. M., and P. F. Reynolds, Jr. (1990). Optimal dynamic remapping of data parallel computations. *IEEE Trans. on Computers* **39**, 206–219.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, USA.

Palmer, R. G. (1989). Neural nets. In D. L. Stein (Ed.), *Lectures in the Sciences of Complexity*, 439–461. Addison-Wesley, Redwood City, CA, USA.

Papadimitriou, C. H., and K. Steiglitz (1982). *Combinatorial Optimization.* Prentice-Hall, Englewood Cliffs, NJ, USA.

Pase, D. M., and A. R. Larrabee (1988). Intel iPSC concurrent computer. In R. G. Babb II (Ed.), *Programming Parallel Processors*, 105–124. Addison-Wesley, Reading, MA, USA.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, USA.

Pearl, J. (1986). Fusion, propagation and structuring in belief networks. *Artificial Intelligence* **29**, 241–288.

Pearl, J. (1987). Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence* **32**, 245–257.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, USA.

Pearl, J. (1989). Probabilistic semantics for nonmonotonic reasoning: a survey. In *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning*, 505–516.

Perrott, R. H. (1987). *Parallel Programming.* Addison-Wesley, Wokingham, England.

Peterson, C. (1990). Parallel distributed approaches to combinatorial optimization: benchmark studies on traveling salesman problem. *Neural Computation* **2**, 261–269.

Peterson, J. L., and A. Silberschatz (1985). *Operating System Concepts.* Addison-Wesley, Reading, MA, USA.

Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science* **13**, 45–60.

Portella, C. F., and V. C. Barbosa (1992). Parallel maximum-flow algorithms on a transputer hypercube. Technical report ES-251/92, COPPE/UFRJ, Rio de Janeiro, Brazil.

Rabin, M., and D. Lehmann (1981). On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proc. of the Annual ACM Symposium on Principles of Programming Languages*, 133–138.

Ramachandran, V., M. Solomon, and M. Vernon (1987). Hardware support for interprocess communication. In *Proc. of the Annual International Symposium* *on Computer Architecture*, 178–188.

Rota, G.-C. (1964). On the foundations of combinatorial theory, I. Theory of Möbius functions. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete* **2**, 340–368.

Rozanov, Y. A. (1969). *Probability Theory: A Concise Course*. Dover, New York, NY, USA.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, 318–362. The MIT Press, Cambridge, MA, USA.

Samlowski, W. (1976). Case grammar. In E. Charniak and Y. Wilks (Eds.), *Computational Semantics*, 65–71. North-Holland, Amsterdam, The Netherlands.

Segall, A. (1983). Distributed network protocols. *IEEE Trans. on Information Theory* **IT-29**, 23–35.

Seitz, C. L. (1985). The cosmic cube. *Comm. of the ACM* **28**, 22–33.

Selman, B. (1985). Rule-based processing in a connectionist system for natural language understanding. Technical report CSRI-168, Department of Computer Science, University of Toronto, Toronto, Canada.

Shen, C.-C., and W.-H. Tsai (1985). A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. on Computers* **C-34**, 197–203.

Simmen, M. W. (1991). Parameter sensitivity of the elastic net approach to the traveling salesman problem. *Neural Computation* **3**, 363–374.

Sinclair, J. B. (1987). Efficient computation of optimal assignments for distributed tasks. *J. of Parallel and Distributed Computing* **4**, 342–362.

Small, S. L., G. W. Cottrell, and M. K. Tanenhaus (Eds.) (1988). *Lexical Ambiguity Resolution*. Morgan Kaufmann, San Mateo, CA, USA.

Spencer, J. (1986). Balancing vectors in the max norm. *Combinatorica* **6**, 55–65.

Spitzer, F. (1971). Markov random fields and Gibbs ensembles. *American Mathematical Monthly* **78**, 142–154.

Stahl, S. (1976). n-tuple colorings and associated graphs. *J. of Combinatorial Theory, Series B* **20**, 185–203.

Stahl, S. (1979). Fractional edge colorings. *Cahiers du C.E.R.O.* **21**, 127–131.

Syslo, M. M., N. Deo, and J. S. Kowalik (1983). *Discrete Optimization Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Takefuji, Y., and H. Szu (1989). Design of parallel distributed Cauchy machines. In *Proc. of the International Joint Conference on Neural Networks*, I-529–532.

Trivedi, K. S. (1982). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA.

van den Bout, D. E., and T. K. Miller (1988). A traveling salesman objective function that works. In *Proc. of the IEEE International Conference on Neural Networks*, II-229–303.

van den Bout, D. E., and T. K. Miller (1989). Graph partitioning using annealed

neural networks. In *Proc. of the International Joint Conference on Neural Networks*, I-521–528.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata*, A. W. Burks (Ed.). University of Illinois Press, Urbana, IL, USA.

Waltz, D. L., and J. B. Pollack (1985). Massively parallel parsing: a strongly interactive model of natural language interpretation. *Cognitive Science* **9**, 51–74.

Welch, J. L., L. Lamport, and N. A. Lynch (1988). A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 28–43.

Wilson, G. V., and G. D. Pawley (1988). On the stability of the travelling salesman problem algorithm of Hopfield and Tank. *Biological Cybernetics* **58**, 63–70.

Wilson, R. J. (1979). *Introduction to Graph Theory*. Longman, London, England.

Winograd, T. (1983). *Language as a Cognitive Process, Vol. 1: Syntax*. Addison-Wesley, Reading, MA, USA.

Wolfram, S. (1984). Universality and complexity in cellular automata. *Physica D* **10**, 1–35.

Wolfram, S. (Ed.) (1986). *Theory and Applications of Cellular Automata*. World Scientific, Singapore.

Wylie, C. R. (1975). *Advanced Engineering Mathematics*. McGraw-Hill, Tokyo, Japan.

Author index

Abu-Mostafa, Y., 6.
Ackley, D. H., 150.
Ahuja, R. K., 151.
Aiso, H., 101.
Akiyama, Y., 101.
Akl, S. G., 48.
Allen, J., 175.
Almasi, G. S., 48.
Andrews, G. R., 72, 198.
Angluin, D., 48.
Aragon, C. R., 150.
Arjomandi, E., 48.
Arlauskas, S., 199.
Attiya, H., 48.
Awerbuch, B., 71–72, 151.

Bacchus, F., 175.
Barbosa, V. C., 7, 16, 72, 101–102, 114–115, 133, 150–151, 175, 198–199, 209, 221, 224.
Bell, G., 48.
Ben-Ari, M., 71, 198.
Berge, C., 16.
Bertsekas, D. P., 48, 72, 114, 199.
Besag, J., 149.
Boeres, M. Claudia S., 102.
Boeres, M. Cristina S., 133, 150–151.
Bondy, J. A., 16.
Bracha, G., 48.
Brandt, R. D., 101.
Braun, M., 101, 114.
Burns, A., 209.
Burr, D. J., 102.

Chandy, K. M., 48, 72.
Chang, E., 72.
Chao, L., 199.
Charniak, E., 175.
Chien, A., 199.
Chin, F., 71.
Chou, C.-T., 48.
Christofides, N., 101.
Collins, N. E., 150.
Cook, S. A., 16.
Cooper, G. F., 175.
Cormen, T. H., 17, 48, 151, 199.
Cottrell, G. W., 175.
Cybenko, G., 199.

Dally, W. J., 199.
Dantzig, G. B., 114.
de Carvalho, L. A. V., 101–102, 114–115.
Deo, N., 199.
Dijkstra, E. W., 48, 72, 198.
Drummond, L. M. de A., 209.
Durbin, R., 102.

Edmonds, J., 199.
Eglese, R. W., 150.
Eizirik, L. M. R., 175.
Even, S., 16, 151.

Fanty, M., 175.
Feldman, Y., 48.
Feller, W., 17.
Felten, E., 150.
Fillmore, C. J., 175.
Fiorini, S., 16, 199.
Fischer, M. J., 48.
Ford, Jr., L. R., 151.
Fox, G. C., 5, 48, 199.
França, F. M. G., 209.
Freitas, L. de P. Sá, 199.
Fulkerson, D. R., 151, 199.
Furmanski, W., 199.

Gafni, E., 16, 48, 72, 133, 150, 210, 221.
Gallager, R. G., 71, 199.
Garey, M. R., 16, 101, 133.
Geiger, D., 175.
Gelatt, C. D., 150.
Geman, D., 150.
Geman, S., 150.
Gentleman, W. M., 198.
Gerla, M., 199.
Gibbons, A., 48.
Goldberg, A. V., 151.
Golden, B. L., 150.
Goldman, K. J., 48.
Goldman, R. P., 175.
Goles, E., 5, 87.
Golub, G. H., 114.
Gonzaga, C. C., 115.
Gottlieb, A., 48.
Greening, D. R., 150.
Griffeath, D., 149, 221.
Grimmett, G. R., 149, 221.
Grötschel, M., 114.
Grunwald, D. C., 199.
Günther, K. D., 199.

Harary, F., 16.
Hassoun, S., 199.
Hecht-Nielsen, R., 6.
Hegde, S. U., 101.
Hellmuth, A. L. H., 209–210.
Henrion, M., 175.
Herskovits, J., 115.
Hertz, J., 6, 133.
Hillis, W. D., 48.
Hinton, G. E., 6, 150.
Hoare, C. A. R., 198, 209.
Hopfield, J. J., 101, 115, 133.
Horwat, W., 199.
Hrycej, T., 175.
Huang, H. K., 199.
Humblet, P. A., 71.

Isham, V., 149.

JáJá, J., 48.
Jen, E., 87.
Jiang, T., 6.
Johnson, D. S., 16, 101, 133, 150.
Johnson, M. A., 48.

Kajiura, M., 101.
Kaplan, J., 199.
Karlin, Samuel, 17, 150.
Karlin, Scott, 150.
Karmarkar, N., 115.
Karp, R. M., 16, 48, 133.
Kawamoto, A. H., 175.
Kermani, P., 199.
Kernighan, B. W., 101.
Khachiyan, L. G., 114.
Kinderman, R., 149.
Kirkpatrick, S., 150.
Kleinrock, L., 17, 199.
Kohonen, T., 133.
Kolawa, A., 199.
Kosko, B., 6.
Kowalik, J. S., 199.
Krogh, A., 6, 133.
Krumme, D. W., 199.

Lamport, L., 48.
Larrabee, A. R., 199.
Laub, A. J., 101.
Lawler, E. L., 151.
Lebowitz, J. L., 87.
Lee, E. Y. S., 199.
Lee, Y. C., 6.
Lehmann, D., 72.
Leighton, F. T., 48.
Leiserson, C. E., 17, 48, 151, 199.
Levy, W. B., 101.
Lima, P. M. V., 72, 101, 133.
Lin, S., 101.
Lovász, L., 114.
Luenberger, D. G., 114.
Lynch, N. A., 48, 72.
Lyzenga, G. A., 48.

Ma, P. R., 199.
Maekawa, M., 72, 198.
Maes, C., 87.
Magnanti, T. L., 151.
Manna, Z., 48.
Martínez, S., 5, 87.
McClelland, J. L., 175.
McGeoch, L. A., 150.
Mead, C., 6, 101, 114.
Mendes, S. B. T., 175.
Metropolis, N., 150.
Miller, T. K., 101.
Misra, J., 48, 72.
Mitra, D., 150.
Mitra, S. K., 101.
Moussouris, J., 150.
Murty, U. S. R., 16.

Nicol, D. M., 199.
Nilsson, N. J., 199.

Oldehoeft, A. E., 72, 198.
Oldehoeft, R. R., 72, 198.
Orlin, J. B., 151.
Otto, S. W., 5, 48, 150.

Palmer, R. G., 6, 133.
Papadimitriou, C. H., 17, 101, 114, 151.
Pase, D. M., 199.
Pawley, G. D., 101.
Pearl, J., 175, 199.
Peleg, D., 71–72.
Perrott, R. H., 198, 209.
Peterson, C., 101.
Peterson, J. L., 71, 198.
Pnueli, A., 48.
Pollack, J. B., 175.
Portella, C. F., 151.

Rabin, M., 72.
Ramachandran, V., 48, 199.
Reed, D. A., 199.
Reynolds, Jr., P. F., 199.
Rivest, R. L., 17, 48, 151, 199.
Romeo, F., 150.
Rosenbluth, A. W., 150.
Rosenbluth, M. N., 150.
Rota, G.-C., 221.
Rozanov, Y. A., 17.
Rumelhart, D. E., 6.
Rytter, W., 48.

Salmon, J. K., 48.
Samlowski, W., 175.
Sangiovanni-Vincentelli, A., 150.
Schevon, C., 150.
Schneider, F. B., 72, 198.
Scholten, C. S., 48.
Schrijver, A., 114.
Schweizer, D., 6.
Segall, A., 48, 209.
Seitz, C. L., 48, 199.
Sejnowski, T. J., 150.
Selman, B., 175.
Shapiro, E., 48.
Shen, C.-C., 199.
Silberschatz, A., 72, 198.
Simmen, M. W., 102.
Sinclair, J. B., 199.
Small, S. L., 175.
Snell, J. L., 149.

Snir, M., 48.
Solomon, M., 199.
Song, P., 199.
Speer, E. R., 87.
Spencer, J., 6.
Spira, P. M., 71.
Spitzer, F., 149.
Stahl, S., 16, 72, 199, 221.
Steiglitz, K., 17, 101, 114, 151.
Sweet, J. L., 101.
Syslo, M. M., 199.
Szeliski, R., 102.
Szu, H., 101.

Takefuji, Y., 101.
Tanenhaus, M. K., 175.
Tank, D. W., 101.
Tardos, E., 151.
Tarjan, R. E., 151.
Taylor, H. M., 17, 150.
Teller, A. H., 150.
Teller, E., 150.
Ting, H. F., 71.
Totty, B., 199.
Toueg, S., 48.
Trivedi, K. S., 17.
Tsai, W.-H., 199.
Tsitsiklis, J. N., 48, 72, 114.
Tsuchiya, M., 199.
Tuttle, M. R., 48.

Ulam, S., 87.

van den Bout, D. E., 101.
van Loan, C. F., 114.
Vecchi, M. P., 150.
Venkataraman, K. N., 199.
Verma, T., 175.
Vernon, M., 199.
von Neumann, J., 87.

Walker, D. W., 48.
Waltz, D. L., 175.
Wang, Y., 101.
Warmuth, M. K., 48.
Welch, J. L., 48.
Williams, R., 199.
Williams, R. J., 6.
Wills, S., 199.
Willshaw, D., 102.
Wilson, G. V., 101.
Wilson, R. J., 16, 199.
Winograd, T., 175.
Wolfram, S., 87.
Wylie, C. R., 101, 114.

Yamashita, A., 101.
Yuille, A., 102.

Subject index

A^* algorithm, 190–191, 199.
heuristic function for the, 191.
admissible, 191.
Algorithm,
A_Alg, 50.
A_Alg(*Alpha*), 52.
A_Pif, 32.
Diff, 45.
Gsr, 39.
S_Alg, 50.
Sas, 55.
Seq_Analog_Hopfield, 94.
Seq_Annealed_Bayesian, 165.
Seq_Annealed_MRF, 145.
Seq_Bayesian, 160.
Seq_Binary_Hopfield, 123.
Seq_Cellular_Automaton, 81.
Seq_Least_Squares, 110.
Seq_MRF, 139.
Seq_Simulated_Annealing, 143.
Ser, 60.
S_Pif, 31.
Task(t), 181.
analog VLSI, 3, 6.
artificial intelligence, 149.
associative memory, 125–129, 133.
binary Hopfield neural network as an, 125–129, 133.
capacity of a, 128.
crosstalk term in a, 128.
Hamming distance between patterns in an, 125–127.
pattern in an, 126–129.
atomic action, 31, 39, 179.
automaton network, 3–7.
computer simulation of an, 3.
distributed parallel simulation of an, 3, 66–72.
FC, 4.
FC updating schedule for an, 4–5.
Fully Concurrent; *see* automaton network, FC.
node state in an, 4.
initial, 4.
Partially Concurrent; *see* automaton network, PC.
PC, 4–5.
PC updating schedule for an, 4–5.
state of an, 4.
updating function of an, 3.
updating rule of an, 4–5.
updating schedule for an, 4.

back-propagation learning algorithm, 4, 6.
Bayesian network, 153–175.
as a GRF, 157–159, 163–164, 175.

as a PC automaton network, 159–167, 175.
conditional probabilities in a, 154.
extreme, 175.
distributed parallel simulation of a, 161–163, 166–167, 175.
message types in the, 161–162.
with simulated annealing, 166–167, 175.
d-separation in a, 175.
energy function of a, 158–159, 163–164.
input to a, 155.
intractability of solving a, 156, 175.
joint posterior distribution in a, 156.
maximizing the, 163–164.
posterior probabilities in a, 155.
potentials of a, 158.
prior probabilities in a, 154.
random variables in a, 153.
child, 154.
evidence, 155.
initial values of the, 159.
mate, 154.
parent, 154.
sequential simulation of a, 159–160, 164–165.
with simulated annealing, 164–165.
stochastic simulation of a, 156, 175.
as an instance of Gibbs sampling, 157–159, 175.
variations of the, 163, 175.
uncertain evidential reasoning with a, 154, 168.
underlying undirected graph of a, 159–160.
Bayes's formula, 14.
Bernoulli trials, 128.
binary relation, 26.
acyclic, 26.
irreflexive, 27.
r-transitive closure of a, 75.
symmetric, 10.
transitive, 27.
transitive closure of a, 27.
biology, 3.
Boltzmann machine, 147–150.
as a GRF, 147.
as related to a binary Hopfield neural network, 147–148.
energy function of a, 147.
potentials of a, 147.
temperature in a, 147–148.
Boltzmann's constant, 141.

causal chain in a distributed computation, 29.
cellular automaton, 75–87.
as an FC automaton network, 79–82.
cylindrical, 76.
deterministic, 76.
distributed parallel simulation of a, 81–82.
finite, 76.
fixed point of a, 77.
infinite, 77.
limit cycle of a, 76.
length of a, 77.
node state in a, 76.
initial, 79.
nondeterministic, 77.
one-dimensional, 77–85.
categories of a, 77–81, 87.
probabilistic, 85–87.
sequential simulation of a, 79–81.
state of a, 76.
as a string in the one-dimensional case, 83.
underlying lattice of a, 75.
updating function of a, 76.
compositions of the, 83.
probabilistic, 85.
radius of the, 76.

chip firing problem, 4, 6.
clique, 10.
clock,
global, 23.
local, 24.
combinatorial optimization,
k-optimality in, 97.
combinatorial optimization problem, 12.
decision-problem form of a, 12.
NP-hard, 13.
Communicating Sequential Processes; *see* CSP.
communication,
asynchronous, 181.
blocking, 181.
nonblocking, 181.
synchronous, 181.
communication channel,
bidirectional, 23.
capacity of a, 24, 182.
infinite, 24, 182.
positive, 182.
zero, 182.
communication demand of a, 188.
FIFO, 40.
First In, First Out; *see* communication channel, FIFO.
unidirectional, 180.
communication link, 187.
capacity of a, 188.
communication processor, 196, 199.
complex system, 3–5.
complexity,
O notation for, 12.
complexity of a distributed parallel algorithm, 28–30.
communication, 28.
time, 28.
asynchronous, 28–29.
synchronous, 28–29.
complexity of a sequential algorithm, 12.
complexity theory, 11–13, 16.
computer networks, 22, 199.
computer science, 3–4, 12, 141.
concurrency,
in automaton networks, 5.
in binary Hopfield neural networks,
as associative memories, 128–129.
used to solve MNCP, 131–132.
in MRF's, 140.
concurrent program, 22.
concurrent programming, 39–40, 71, 181.
consistent global state; *see* global state.
content-addressable memory; *see* associative memory.
convex function, 104.
Corollary,
4.2, 62.
C.3, 212.
C.4, 212.
D.2, 224.
CSP, 181–182, 198, 201, 209.

deadlock, 22.
in message buffering, 183–187.
in Occam programs, 203.
in resource sharing, 57.
in routing, 196–198.
decision problems, 12–13.
algorithms for,
efficient, 12.
exponential-time, 12.
polynomial-time, 12.
class *NP* of, 12–13.
class *P* of, 12–13.
NP-complete, 12–13, 16.
NP-hard, 13.
Dining Philosophers Problem; *see* DPP.
directed graph, 10–11.
acyclic, 10.
sink decomposition of an, 11, 16.
sink in an, 11.

source in an, 11.
directed cycle in a, 10.
directed path in a, 10.
edge set of a, 10.
symmetric, 10.
node set of a, 10.
strongly connected, 10.
underlying undirected graph of a, 10.
discrete random variable; *see* random variable.
distributed algorithm, 21–32, 48.
initiating a, 22.
notation, 30–32.
distributed parallel algorithm, 19.
distributed snapshot; *see* global state.
distributed system, 22.
anonymous, 32, 48.
knowledge of size in a, 32, 48.
message-driven, 22.
point-to-point, 23.
neighbors in a, 23.
reactive, 22.
distribution, 13.
Boltzmann-Gibbs, 136.
joint, 15.
DPP, 56–57, 72.
generalized, 57, 72.
dynamic system,
discrete, 3.
Lyapunov function of a, 91.

economics, 3.
elastic-net method, 102.
engineering, 103.
events in a distributed computation, 25.
causality among, 26.
concurrent, 28.
consecutive, 34.
future of, 28.
internal, 26.
partial order on, 27.
past of, 28.
set of, 25.
partition of the, 35.
total order on, 33.
consistent with a partial order, 33.
events in probability theory, 13.
independent, 14.

feasible point, 11.
firing squad problem, 4, 6.
flit, 198.
flow-control digit; *see* flit.
forest, 9.
spanning, 10.

generalized Fibonacci series, 85.
Gibbs random field; *see* GRF.
global state, 33–41, 48.
earlier, 37.
earliest, 37.
future of a, 37.
past of a, 37.
recording a, 37–41, 48.
state of a channel in a, 33.
state of a processor in a, 33.
Global state recording; *see* Gsr.
graph theory, 9–11, 16.
GRF, 135–151, 216–221.
as related to an MRF, 136–137, 216–221.
energy function of a, 136–137.
potentials of a, 136.
temperature in a, 136.
Gsr, 37.
guarded command, 40, 48, 180, 198.
command in a, 180.
guard in a, 180.

Hopfield neural network, 89–102, 119–133, 192–195.
analog, 89–102, 192–195.
a learning algorithm for an, 101.
as a dynamic system, 91.
as an FC automaton network, 92–96.

avoiding the gradient-descent behavior of an, 101.
capacitance of a neuron in an, 90.
differential equations for an, 89.
distributed parallel simulation of an, 95–96, 101.
electric-circuit model for an, 90.
energy function of an, 90.
under high gains, 91–92.
external input to a neuron in an, 90.
gain of a neuron in an, 89.
input to a neuron in an, 89–90.
neuron in an, 89.
neuron state in an, 89.
initial, 92.
potential of a neuron in an, 89.
resistance of a neuron in an, 90.
sequential simulation of an, 92–95.
sigmoid transfer function for an, 89.
state of an, 90.
stable, 91.
synaptic strengths in an, 90.
threshold of a neuron in an, 89.
threshold potential of a neuron in an; *see* Hopfield neural network, analog, threshold of a neuron in an.
binary, 119–133.
as a PC automaton network, 122–125.
as related to a Boltzmann machine, 147–148.
distributed parallel simulation of a, 124–125, 133.
nondeterministic, 124–125.
energy function of a, 120.
external input to a neuron in a, 119.
Hebb rule for learning in a, 127–128.
input to a neuron in a, 119.
neuron in a, 119.
neuron state in a, 119.
initial, 123.
potential of a neuron in a, 119.
sequential simulation of a, 122–124.
nondeterministic, 123–124.
sign transfer function for a, 125.
state of a, 122.
stable, 122.
step transfer function for a, 119.
synaptic strengths in a, 119.
threshold of a neuron in a, 120.
threshold potential of a neuron in a; *see* Hopfield neural network, binary, threshold of a neuron in a.
Hopfield-Tank model for combinatorial optimization, 97–101.

infinitely-often condition, 5.
initial-value problem, 92.
Euler's method for an, 93.
Runge-Kutta method for an, 94.
input/action pair, 30.
intractable problems; *see* combinatorial optimization problem, *NP*-hard; decision problems, *NP*-complete; *and* decision problems, *NP*-hard.

knowledge representation, 174–175.

lattice, 75.
  multidimensional, 75.
    cylindrical, 76.
    periodic boundary conditions for a, 75.
Lemma,
  C.1, 211.
  C.5, 214.
  C.6, 215.
  C.7, 216.
lexical disambiguation, 168–175.
  Bayesian networks for, 168–175.
  connectionist approaches to, 168, 175.
  probabilistic approaches to, 168–175.
  traditional approaches to, 168, 175.
  verb cases in, 168.
lexicographic product of undirected graphs, 10.
linear program, 111.
  analog neural network to solve a, 112–115.
  dual, 111.
  primal, 111.
  standard form of a, 111.
linear programming, 111–115.
  ellipsoid method for, 114.
  feasible-direction methods for, 114–115.
    analog neural network based on, 115.
  interior-point methods for, 114–115.
linear programming problem; *see* linear program.
linear system, 103.
  analog neural network to solve a, 103–111.
    as a dynamic system, 107–108.
    as an FC automaton network, 108–111.
    capacitance of some neurons in the, 105.
    distributed parallel simulation of the, 110–111.
    electric-circuit model for the, 105.
    equations for the, 105.
    external input to a neuron in the, 105.
    neuron in the, 105.
    neuron state in the, 105.
      initial, 108.
    resistances of neurons in the, 105.
    sequential simulation of the, 108–110.
    state of the, 107.
      stable, 107.
    synaptic strengths in the, 107.
  direct methods to solve a, 104.
  Gaussian elimination to solve a, 104.
  Gauss-Seidel method to solve a, 104.
  iterative methods to solve a, 104.
  least-squares formulation of a, 104–105.
  number of solutions of a, 103–104.
  parallel methods to solve a, 114.

Markov chain, 16.
  homogeneous, 16.
  one-step transition probability in a; *see* Markov chain, transition probability in a.
  reversible, 16.
  stationary distribution of a, 16.
  transition probability in a, 16.
Markov random field; *see* MRF.
matching, 10.
  complete, 10.
  maximal, 10.
matrix,
  Hessian, 104.
    semi-positive definite, 104.
  linearly independent columns of a, 104.
  Moore-Penrose conditions on a, 105.
  pseudo-inverse of a, 105.
  rank of a, 104.
  square,
    characteristic polynomial of a, 84.
    determinant of a, 84.
    eigenvalues of a, 84.
    identity, 84.
    inverse of a, 108.
      analog neural network to compute the, 108.
    trace of a, 83.
  transpose of a, 104.
maximum flow in a network, 150–151.
  distributed parallel algorithms for, 151.
  sequential algorithms for, 151.
medical sciences, 168.
message,
  ack, 44, 51.
  comp_msg, 38, 44, 51, 57.
  inf, 29.
  marker, 38.
  safe, 51.
  startup, 52, 59.
  stop, 68, 70.
message buffering, 182–187, 199.
  concurrency-optimal criterion for, 186–187.
  intractability of optimal, 184.
  relay tasks for, 183.
  space-optimal criterion for, 184–185.
message passing, 22.
messages in transit, 33, 183.
  bounds on the number of, 183.
Metropolis algorithm, 141, 150.
Minimum Node Cover Problem; *see* MNCP.
MNCP, 129–133, 149–151.
  a general formulation for problems like, 133.
  formulation as an unconstrained optimization problem for, 130, 133.
  heuristic with performance bound for, 130.
  intractability of, 129, 133.
  use of a binary Hopfield neural network for, 130–133.
  use of a Boltzmann machine for, 149–151.
    maximum-flow computation in the, 150–151.
  use of the Hopfield-Tank model for, 130–133.
Möbius inversion formulas, 216, 221.
model of computation,
  Parallel Random Access Machine; *see* model of computation, PRAM.
  PRAM, 23, 48.
  RAM, 23.
  Random Access Machine; *see* model of computation, RAM.
model of distributed computation, 23–25, 48.
  asynchronism in a, 23.
  asynchronous, 23–24.
  fully asynchronous; *see* model of distributed computation, asynchronous.
  fully synchronous; *see* model of distributed computation, synchronous.
  synchronism in a, 23.
  synchronous, 23–24.
  timing in a, 23.
MRF, 135–151.
  as a PC automaton network, 138–140, 144–146.
  as related to a GRF, 136–137, 216–221.

distributed parallel simulation of an, 139–140, 145–146, 150.
with simulated annealing, 145–146.
generalized Markovian dependency in an, 136.
Gibbs sampling on an, 137, 150.
ergodicity of the, 146–147, 150.
simulated annealing in the, 143–144.
cooling schedule for, 144.
updating round of, 137.
homogeneous neighborhood in an, 135.
intractability of solving an, 137.
local characteristics of an, 137.
positivity requirement on an, 136, 149–150.
random variables in an, 135.
initial values of the, 138.
sequential simulation of an, 138–139, 144–145.
with simulated annealing, 144–145.
multigraph, 10.
mutual exclusion, 57.

natural language processing, 168–175.
ambiguities in, 168–175.
lexical, 168–175.
structural, 168.
grammatical categories in, 168–170.
semantic analysis in, 169–174.
syntactic analysis in, 169–174.
ungrammatical sentences in, 169.
word senses in, 168–174.
neighborhood constraints, 56–57.
neighborhood-constrained systems, 57.
neural network, 3, 5–6.
feed-forward connections in a, 4.
learning in a, 3.

node cover, 10.

Occam, 201–209.
CHAN variables in, 202.
FC automaton network simulation in, 209.
PC automaton network simulation in, 209.
positive channel capacity in, 203–206.
process in, 201–203.
alternative, 203.
termination of an, 203.
parallel, 202.
priority, 202.
termination of a, 202.
primitive, 201–202.
assignment, 201.
input, 202.
output, 202.
termination of a, 202.
replication of a nonprimitive, 203.
sequential, 202.
termination of a, 202.
programming methodology for, 206–210.
message buffering in a, 207.
message demultiplexing in a, 207–208.
message multiplexing in a, 207.
message routing in a, 207.
zero channel capacity in, 202–203.
operating systems, 22, 56, 71, 183.
optimization problem, 12.
orientation of a graph; *see* orientation.
orientation, 10.
acyclic, 11.
downstream neighbor in an, 59.
upstream neighbor in an, 59.

parallel processing, 5.

systems for, 3, 21–23.
agents in, 21–22.
levels of parallelism in, 21–22.
parallel program, 21.
physics, 3.
Pif, 29.
precedence graph, 26–28.
cut in a, 35.
probability,
conditional, 14.
probability distribution; *see* distribution.
probability theory, 13–17.
procedure,
COMPUTE$_i$, 69, 71.
INITIALIZE$_i$, 82, 96, 111, 124, 140, 146, 162, 167.
NOTIFY$_i$, 82, 96, 111, 124, 140, 146, 162, 167.
UPDATE$_i$, 82, 96, 111, 124, 140, 146, 162, 167.
process, 21–22.
process allocation, 23.
processor, 22.
physical, 21–22, 187.
speed of a, 187–188.
state of a, 25.
final, 26.
initial, 26.
Propagation of information with feedback; *see* Pif.
pulse, 4, 24.

random variable, 15.
average of a, 15.
expected value of a, 15.
resource sharing, 56, 71–72.
routing, 196–199.
circuit-switching, 197.
e-cube, 197, 199.
flow control and, 198.
function, 196.
store-and-forward, 196.
structure, 187.
structured buffer pools in, 197.
virtual cut-through, 198–199.
wormhole, 198–199.
virtual channels in, 198.
virtual links in, 198.

sample space, 13.
Scheduling by alpha synchronization, 54.
Scheduling by edge reversal, 59.
scheduling by edge reversal, 56–66, 72, 211–216.
concurrency measure for, 62.
intractability in, 66, 72.
optimality of, 65–66, 72.
period in, 62.
period of orientations in; *see* scheduling by edge reversal, period in.
periodic orientation in, 62.
properties of, 61–66, 72.
reachability among orientations in, 223.
reachability equations in, 224.
schedule of orientations in, 223.
simulated annealing, 140–143, 150.
as related to the physical annealing, 141–142.
convergence of, 150.
experimental evaluations of, 150.
parallel versions of, 150.
short-term properties of, 150.
temperature in, 141.
final, 142.
initial, 142.
number of iterations for a same, 142.
reduction rate for the, 142.
stable property of a distributed computation, 42.
starvation, 22.
stationary process, 16.
ergodicity of a, 146.
statistical physics, 136.
stochastic process, 15–16.
discrete-parameter, 15–16.

discrete-state, 15–16.
discrete-time, 15.
state space of a, 15.
substrate distributed parallel computation, 37.
synchronizer, 49–56, 71.
Alpha, 51.
Beta, 55.
communication complexity to initialize a, 50.
Gamma, 56.
per-pulse asynchronous time complexity of a, 50.
per-pulse communication complexity of a, 50.
processor safety in a, 50.
time complexity to initialize a, 51.
synchronous pulse; *see* pulse.
system of linear algebraic equations; *see* linear system.
system state, 33.

task, 179.
processing demand of a, 188.
task allocation, 187–195, 199.
allocation function for, 187.
as data partitioning, 187.
dynamic, 188.
intractability of optimal, 189, 199.
on a hypercube, 189, 199.
static, 188.
use of an analog Hopfield neural network for, 192–195.
use of neural networks for, 199.
use of the A^* algorithm for, 190–192.
task rendez-vous, 181.
template simulation algorithms, 66–72.
a generic architecture for, 66–67.
computing statistics in, 66–67.
for FC automaton networks, 67–69.
for PC automaton networks, 69–71.
maintaining logs in, 66.
termination,
global, 41.
termination detection, 41–48.
for diffusing computations, 43–48.
for general distributed computations, 41–43.
Theorem,
3.1, 29.
3.2, 40.
3.3, 46.
4.1, 61.
4.3, 65.
4.4, 65.
5.1, 83.
6.1, 90.
6.2, 92.
6.3, 100.
7.1, 107.
8.1, 121.
8.2, 130.
9.1, 136.
9.2, 137.
9.3, 143.
9.4, 146.
10.1, 155.
10.2, 156.
10.3, 158.
A.1, 184.
A.2, 194.
C.2, 211.
D.1, 224.
total probability formula, 14.
transputers, 204, 206, 209.
Traveling Salesman Problem; *see* TSP.
tree, 9.
spanning, 10.
TSP, 96–102.
Christofides heuristic for, 97, 101.
feasible tour for, 96.
intractability of, 96, 101.
Lin-Kernighan heuristic for, 97, 101.
triangle inequality for, 96.

use of an analog Hopfield neural network for, 97–102.
use of the Hopfield-Tank model for, 97–102.
Turing machine, 12.

undirected graph, 9–10, 16.
completely connected, 10.
connected, 9.
connected components of an, 9.
cycle in an, 9.
Eulerian, 10.
Hamiltonian, 10.
simple, 10.
degree of a node in an, 9.
diameter of an, 9.
edge coloring in an, 10.
edge multicoloring in an, 10, 16.
edge set of an, 9.
Eulerian, 10.
Hamiltonian, 10.
independent set in an, 5.
neighbors in an, 3.
node coloring in an, 10.
node multicoloring in an, 10, 16.
node set of an, 9.
path in an, 9.
shortest, 9.